VICTOR VALERIU PATRICIU

Sisteme de operare pentru mini și microcalculatoare

VICTOR VALERIU PATRICIU

Sisteme de operare pentru mini și microcalculatoare

EDITURA MILITARĂ, BUCUREȘTI, 1992

În cadrul lucrării se fac referiri la următoarele produse hard și soft care sînt mărci înregistrate:

— UNIX — marca AT&T Bell Laboratories;
— VAX și VMS — mărci Digital Equipement Corporation;
— MS-DOS — marca Microsoft Corporation;
— PC-DOS — marca IBM;
— IBM-PCXT/AT — marca IBM;
— OS/2 — marca IBM.

Redactor MIȘU RĂILEANU
Tehnoredactor: MARIA NICOLAE

Bun de tipar 03.08.1992. Apărut 1992.
Coli tipar 23.B 10591.

Tiparul executat sub comanda nr. 19 la I. P. „Filaret", str. Fabrica de chibrituri nr. 9—11, București
România

ISBN 973-32-0309-2

CUVÎNT ÎNAINTE

În urmă cu numai cinci decenii se năşteau primele calculatoare electronice sub imperiul necesităţilor militare ale celui de-al doilea război mondial. Un lanţ de dezvoltări explozive ale ştiinţei (fizica, chimia, electronica, microelectroniça, mecanica fină) au marcat evoluţia spectaculoasă a calculatoarelor prezente azi în toate domeniile vieţii noastre. Dacă în urmă cu ani, informatica părea o religie exotică, practicată în nişte lăcaşe ciudate şi îndepărtate, numite centre de calcul, unde se întîmplau lucruri misterioase pentru puterea de înţelegere a majorităţii oamenilor, astăzi relaţiile om-calculator au devenit firesti datorită implicării tot mai adînci a calculatoarelor nu numai în ştiinţă, economie, proiectare, dar şi în viaţa de toate zilele.

Calculatoarele electronice au devenit astăzi un instrument de lucru la îndemîna unor categorii sociale diferite ca pregătire şi orientare profesională, de la simplul elev de gimnaziu la profesorul universitar, de la inginerul din producţie la patronul de firmă.

Calculatorul este implicat direct sau indirect în viaţa fiecărui locuitor al planetei: dacă nu lucrează direct cu el, adică nu-l foloseşte ca mijloc de producţie sau instrument de lucru, atunci apelează la el cel puţin pentru a transmite poştă electronică, pentru-a-şi rezerva locuri la hoteluri, spectacole sau în avioane, acceptă facturile de telefon sau electricitate scoase pe calculator sau beneficiază de evidenţa ţinută de spitale sau policlinici în fişiere sau baze de date.

Orice calculator poate fi privit ca un ansamblu de echipamente electronice (hardware) *și programe* (software). *Dacă echipamentele sînt cele care conferă, prin tehnologiile de vîrf implementate, performanțele de viteză de prelucrare și capacitate de stocare a informațiilor, software-ul este cel care face calculatoarele accesibile utilizatorilor, inteligente și capabile să rezolve o multitudine de probleme. O parte din programe alcătuiesc* soft-ul de bază, *care se achiziționează de obicei o dată cu calculatorul, restul fiind constituit din* soft-ul de aplicație, *cumpărat de la firme specializate sau produs de utilizator.*

Unul din elementele componente cele mai importante ale software-ului îl constituie sistemul de operare. *Acesta poate fi văzut ca o colecție organizată de programe care sînt absolut necesare pentru a face calculatorul utilizabil. Două sînt obiectivele fundamentale ale unui sistem de operare:*

— gestiunea resurselor (procese, procesoare, periferice, memorie, fișiere, programe);

— realizarea unei interfețe cît mai prietenoase cu utilizatorii, prin intermediul căreia aceștia să poată activa programe, manevre periferice și fișiere, dimensiona componentele sistemului.

Scopul acestei lucrări este de a prezenta aspectele constructive și de utilizare a unora dintre cele mai răspîndite sisteme de operare.

Capitolul 1 face o scurtă incursiune în problematica generală a sistemelor de operare. Se prezintă principalele noțiuni care vor fi utilizate în celelalte capitole ale cărții, analizîndu-se sarcinile fundamentale îndeplinite de sistemele de operare: coordonarea și planificarea proceselor, gestiunea intrărilor/ieșirilor, gestiunea sistemului de fișiere și a memoriei interne.

Capitolul 2 este consacrat prezentării UNIX — unul din cele mai populare sisteme de operare, cu o largă răspîndire în special în mediul universitar. Este un sistem capabil să lucreze în time-sharring, iar sursa popularității lui fără precedent o constituie portabilitatea sa. El este disponibil pe o gamă largă de calculatoare, începînd cu cele compatibil IBM-PC și pînă la minicalculatoarele din familia PDP-11 sau VAX. El tinde să devină sistemul de operare universal al anilor următori, așa cum limbajul de programare C a devenit limbajul universal al zilelor noastre.

În capitolul 3 se face o analiză a principalelor aspecte constructive și de utilizare a sistemului de operare VMS de pe minicalculatoarele din familia VAX, pe 32 de biți.

Capitolul 4 prezintă MS-DOS, cel mai folosit sistem de operare de pe calculatoarele personale compatibile IBM-PC. Prezentarea făcută vizează atît modul în care utilizatorul poate dialoga cu acest sistem cît și unele aspecte mai generale privind organizarea internă a MS-DOS, a sistemului sau de fișiere și a modului de gestiune a memoriei interne.

Lucrarea se adresează atît informaticienilor, care pot găsi unele detalii privind principalele sisteme de operare folosite la calculatoarele existente în țara noastră, cît mai ales utilizatorilor neinformaticieni, care pot găsi în carte un sprijin în învățarea lucrului pe mini și microcalculatoare. Ea se dorește un instrument util, accesibil celor cu experiență redusă în folosirea calculatoarelor, care să le permită să dialogheze cu sistemele de operare. Materialele cuprinse în lucrare au constituit suportul de curs postuniversitar de Sisteme de operare ținut în ultimii ani în Academia tehnică militară.

Fără a se substitui manualelor specifice de utilizare, cartea pleacă de la numeroasele lucrări de autor sau de firmă, enumerate în bibliografia selectivă indicată, constituindu-se, sperăm, într-un sprijin real pentru cei care învață singuri să folosească mini și microcalculatoarele.

AUTORUL

Capitolul 1

STRUCTURA SISTEMELOR DE OPERARE

Se poate considera că elementul cel mai important al unui calculator electronic îl constituie programele sale — ansamblul cunoscut sub numele de *software*. În lipsa acestuia calculatoarele nu ar putea memora, prelucra sau regăsi informația, nu ar putea realiza multitudinea de aplicații, de la simplele jocuri pe calculator pînă la programele complexe utilizate în tehnica aerospațială.

Programele unui calculator pot fi împărțite în două categorii: *programe sistem*, care gestionează operațiunile propriu-zise ale calculatorului și *programe de aplicație*, care rezolvă probleme ale utilizatorilor. Partea cea mai importantă din programele sistem o constituie *sistemul de operare*, adică acea parte care gestionează resursele calculatorului, furnizînd suportul software pe care pot fi scrise programele de aplicație.

1.1. INTERACȚIUNEA HARDWARE-SOFTWARE

Cu cîțiva zeci de ani în urmă, la începutul dezvoltării calculatoarelor, programatorii se confruntau direct cu complexitatea hardware-ului, trebuind să construiască aplicațiile direct în limbajul calculatorului, sub forma unor succesiuni de biți. Evoluția ulterioară a calculatoarelor a constat în adăugarea de nivele succesive de software la hardware-ul existent, astfel încît administrarea acestuia și programarea lui să se facă din ce în ce mai simplu. Putem spune că în momentul de față utilizatorul este confruntat cu o *mașină virtuală*, mult mai ușor de înțeles și de programat.

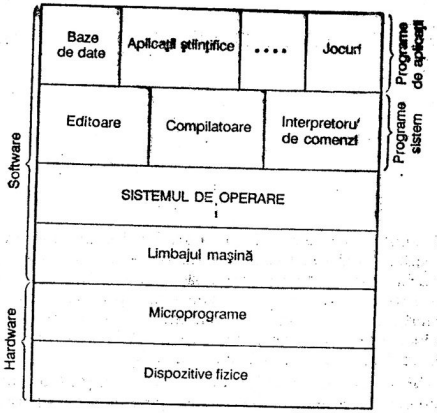


Fig. 1.1. Interacțiunea hard — soft.

În figura 1.1. sînt ilustrate diferitele niveluri care alcătuiesc un calculator.

La baza structurii pe niveluri este hardware-ul, format la rîndul său din mai multe părți. În primul rînd este vorba de *dispozitivele fizice* alcătuite din circuite integrate și cablaje care constituie unitatea centrală, memoria sau cuploarele corespunzătoare configurației.

Următorul nivel consta dintr-un software primar care controlează aceste dispozitive și oferă o interfață unitară nivelului următor. Acest software, numit *microprogram*, este de obicei localizat într-o memorie de tip ROM. El reprezintă de fapt un interpretor al instrucțiunilor în limbaj mașină (spre exemplu ADD, JUMP, MOVE etc.) pe care le traduce într-o serie de pași mici ce permit execuția acestora. Acest ansamblu de instrucțiuni pe care le interpretează microprogramele constituie *limbajul mașină* care, contrar celor crezute de mulți utilizatori, nu face parte

din mașina „reală", ci constituie un nivel adiacent. Pe o serie de calculatoare însă implementarea microprogramului nu se face printr-un software în ROM, ci direct în hardware.

Limbajul mașină conține în mod tipic pînă la 200 de instrucțiuni, grupate în mai multe categorii: aritmetice, de comparare, pentru deplasarea informației în calculator, de intrare/ieșire.

Una din funcțiile de bază ale *sistemului de operare* este de a ascunde această complexitate a hardware-ului și de a furniza programatorului un set de instrucțiuni mult mai accesibil și degrevat de detaliile pentru comanda dispozitivelor fizice.

Deasupra nivelului sistemului de operare se află restul softwareului: compilatoarele, interpretoarele, interpretorul de comenzi, editoarele de texte și alte programe similare. Deși ele nu fac parte propriu-zis din sistemele de operare, în accepțiunea strictă, ele sînt furnizate de către firma producătoare împreună cu acesta.

Deosebirea esențială dintre aceste programe și sistemul de operare este însă următoarea: sistemul de operare este acea porțiune a software-ului care se execută în *mod supervizor*, cu privilegii extinse asupra memoriei și a celorlalte resurse. Compilatoarele, editoarele și celelalte componente ale programelor sistem se execută în *mod utilizator*, cu privilegii reduse, ca orice alt program de aplicații.

Pe nivelul cel mai de sus al structurii indicate se află programele utilizator, cele care rezolvă diferite probleme particulare din domenii variate: economic, științific, de conducere, jocuri etc.

1.2 CE ESTE UN SISTEM DE OPERARE

Deși mulți utilizatori ai calculatoarelor au experiență în lucrul cu sistemele de operare, le este greu să precizeze care sînt funcțiile unui sistem de operare. Pentru a putea explica aceste funcții le vom grupa în două mari categorii:

1) *Sistemul de operare este o mașină virtuală (extinsă)*

Așa cum am arătat arhitectura — setul de instrucțiuni, memoria, structura sistemului de intrare/ieșire — multor

calculatoare la nivelul limbajului mașină este primitivă și greoaie, creînd dificultăți în special în cazul intrărilor/ieșirilor. Drept exemplu să considerăm o unitate de floppy-disc utilizată pe IBM-PC sau alte calculatoare personale care folosește un controler NEC PDP 765. Acest integrat are 16 comenzi, fiecare fiind specificată prin încărcarea unui număr de 1-9 octeți în registrul perifericului. Aceste comenzi sînt pentru citire sau scriere de date, poziționare capete, formatare.

Cele mai utilizate comenzi sînt READ și WRITE, fiecare necesitînd un număr de 13 parametri împachetați în cei 9 octeți. Ei specifică adresa blocului disc accesat, numărul de sectoare pe pistă etc. La sfîrșitul operației de intrare/ieșire integratul furnizează 23 de stări și erori împachetate în 7 octeți. În acest caz programatorul lucrului cu floppy-discul trebuie în acest fel să cunoască o serie de detalii reale (inclusiv dacă motorul este pornit sau oprit). În cazul discului Winchester programarea este și mai complexă. De aceea, utilizînd sistemul de operare, programatorul operează cu fișiere, pe care le deschide sau închide, fișiere în care face simple scrieri sau citiri de articole. Este rolul sistemului de operare de a traduce aceste operații de nivel înalt în operații reale care țin seama de detaliile de programare a perifericelor respective. Este doar un exemplu care indică una din funcțiile majore ale sistemelor de operare, de a realiza o mașină virtuală ori extinsă, ușor de înțeles și de programat de către utilizator.

2) *Sistemul de operare este administratorul resurselor*

Calculatoarele moderne constau din procese, memorii, discuri, terminale, imprimante etc.; sistemul de operare este cel care gestionează toate aceste resurse. Sarcina sa este de a aloca în mod controlat resursele amintite, pe rînd, programelor care sînt în competiție pentru a le ocupa.

Să presupunem că 3 programe se execută pe același calculator, în același timp și doresc să listeze simultan informații pe aceeași imprimantă. Dacă acest lucru ar fi permis fără controlul sistemului de operare, liniile de imprimantă corespunzătoare celor 3 programe se vor întrepătrunde pe listing, făcîndu-l inutilizabil; de aceea sistemul de operare va trebui să creeze cîte un fișier disc pentru

fiecare program unde se vor memora informațiile destinate listării. Apoi tot sistemul de operare va avea grijă să listeze la imprimantă, pe rînd, conținutul acestor fișiere disc.

Alte aspecte foarte importante ale sistemelor care lucrează în multiprogramare îl constituie gestiunea și protecția memoriei, administrarea perifericelor, a fișierelor etc. Sistemul de operare va trebui să memoreze fiecare cerere de resursă, să le ordoneze în fire de așteptare și să le servească după anumite criterii destinate să reducă la maxim timpul de răspuns pentru fiecare program. Tot sistemul de operare va trebui să rezolve anumite cereri conflictuale ale diferitelor programe și utilizatori.

1.3. CONCEPTE ALE SISTEMULUI DE OPERARE

Interfața dintre programele utilizator și sistemul de operare este realizată printr-o mulțime de „instrucțiuni extinse", numite *apeluri sistem*, pe care le furnizează sistemul de operare. Analiza lor permite o înțelegere reală a ceea ce este capabil să execute sistemul de operare. Deși apelurile sistem variază de la un sistem de operare la altul, conceptele de bază cu care operează sînt similare. Din acestea vom aminti două fundamentale: procesul și fișierul.

Procesul este un concept cheie în toate sistemele de operare. El reprezintă un program în cursul execuției. Un proces constă din programul executabil, din datele și stiva asociate programului, din registrele generale, inclusiv contorul de program și pointerul stivei, precum și din alte informații necesare execuției sale. Un concept strîns legat de cel de proces este *multiprogramarea* („time-sharring"); ea sugerează că sistemul de operare decide să stopeze execuția procesului aflat în execuție și să aloce unitatea centrală unui alt proces. În acest fel se dă posibilitatea fiecărui proces să se execute periodic, senzația resimțită de utilizator fiind aceea că programele lor se execută în paralel (în același timp). În realitate acest paralelism este aparent, la un moment dat un singur proces ocupînd unitatea centrală și deci executîndu-se.

Atunci cînd în acest proces de multiprogramare un proces este suspendat temporar, toate informațiile legate de

el — numite context — trebuiesc salvate în vederea reluării lui ulterioare. De exemplu, un proces are deschise mai multe fișiere; poziția exactă în prelucrarea fiecărui fișier trebuie salvată în context, astfel încît la reluarea procesului citirile/scrierile în fișier să se facă în pozițiile corecte.

În multe sisteme de operare, toate operațiile legate de procese, altele decît conținutul spațiului lor de adrese, sînt memorate într-o tabelă a sistemului de operare numită *tabela proceselor* care reprezintă o listă de structuri, cîte una pentru fiecare proces care există.

Ca urmare, un proces (suspendat sau nu) constă din spațiul său de adrese — numit imagine memorie — și dintr-o intrare corespunzătoare în tabela de procese care conține informațiile din registre și alte informații.

Din punctul de vedere al gestiunii proceselor, cele mai importante apeluri sistem sînt de creare și terminare proces. În mod tipic în fiecare sistem de operare există un *interpretor de comenzi* (numit și „shell") care citește comenzi de la terminal, ce reprezintă de obicei apelul unor programe. În aceste cazuri interpretorul trebuie să creeze un proces nou și să lanseze programul apelat. Fiecare proces poate crea la rîndul său alte procese numite *procese fii*, ajungîndu-se astfel la o structură arborescentă (fig. 1.2).

Alte apeluri sistem pentru procese sînt: cerere/eliberare de memorie, așteptare ca un proces fiu să termine execuția, acoperirea programului corespunzător procesului cu altul.

Uneori procesele trebuie să se pună în așteptare pînă cînd alte procese termină o anumită acțiune. Ca urmare trebuie să fie capabile să trimită sau să recepționeze *semnale* la/de la alte procese. Sosirea unui semnal pentru un proces face ca acesta să fie suspendat temporar, cu salvarea contextului și să fie lansată în execuție o procedură specială

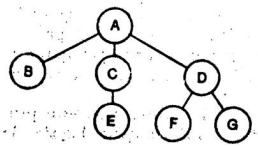


Fig. 1.2. Structura arborescentă a proceselor

de tratare a semnalului. Aceste semnale sînt pentru software ceea ce reprezintă întreruperile pentru hardware. O serie de erori (derutări) detectate hard — cum ar fi instrucțiune inexistentă sau violarea unor zone de memorie interzise — sînt convertite în semnale pentru procese.

Un alt concept fundamental este cel de fișier (o colecție de informații memorate pe un suport magnetic extern, de obicei disc). Una din funcțiile majore ale sistemului de operare o constituie aceea de a furniza un set de apeluri sistem pentru gestiunea sistemului de fișiere. În acest fel sînt ascunse pentru utilizatori particularitățile discurilor și ale altor periferice, prezentînd programatorilor o interfață clară cu un model de fișiere independente de periferice. Apelurile sistem permit crearea, ștergerea, citirea sau scrierea fișierelor.

Majoritatea sistemelor de fișiere folosesc noțiunea de *director* ca o cale de a grupa mai multe fișiere împreună. Un student poate avea de exemplu cîte un director cu programe și fișiere corespunzătoare fiecărui curs, un alt director

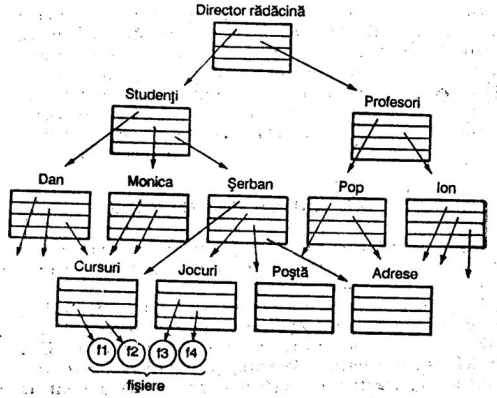


Fig. 1.3. Sistemul de fișiere (exemplu).

pentru poştă electronică şi un altul pentru jocuri. Există apeluri sistem specifice creării sau ştergerii directoarelor, punerii sau scoaterii unui fişier într-un/dintr-un director. Intrările fiecărui director pot fi formate din fişiere sau alte directoare. În acest fel se realizează o ierarhie arborescentă a sistemului de fişiere (figura 1.3).

Spre deosebire de structura arborescentă a proceselor — dinamică şi care durează doar cîteva unităţi de timp —, ierarhia sistemului de fişiere este mult mai stabilă în timp. De asemenea mecanismele de protecţie sînt mult mai complexe în cadrul fişierelor din nevoia de a compartimenta accesul la informaţiile aferente acestora.

Fiecare fişier dintr-o structură arborescentă trebuie specificat printr-o *cale*; ea poate fi absolută, dacă pleacă din directorul rădăcină, sau relativă, dacă pleacă din directorul curent.

O cale este formată dintr-o succesiune de directoare care localizează în mod unic fişierul dorit. În exemplul din figura 1.3 fişierul *f1* se localizează prin calea

/studenţi/Şerban/cursuri/f1.

În general fişierele şi directoarele sînt protejate la acces prin intermediul unor *biţi de protecţie*. Aceştia indică la fiecare fişier acţiunile permise (citire, scriere, execuţie) pentru fiecare categorie de utilizatori (proprietar, grup, toţi ceilalţi). În acest fel, prin mecanisme mai mult sau mai puţin complexe se limitează accesul utilizatorilor doar la fişierele care le aparţin sau la care li se dă dreptul de citire sau scriere.

1.4. TIPURI DE SISTEME DE OPERARE

Din punct de vedere structural intern, sistemele de operare se împart în mai multe categorii, dintre care vom prelua 4 mai importante:

1) *Sisteme monolitice*. În cazul acestora sistemul de operare este scris ca o colecţie de proceduri, fiecare dintre ele putînd apela oricare alta, în funcţie de necesităţi. Fiecare are definită o interfaţă în termenii parametrilor de intrare şi de ieşire. Pentru a construi codul obiect a unui astfel de program — sistem de operare — se compilează separat

aceste proceduri şi se link-editează împreună într-un singur fişier cod obiect. Esenţial în cazul acestor sisteme de operare este că orice procedură este vizibilă pentru oricare alta.

Dar în unele sisteme de operare monolitice se pot găsi germenii unor structuri incipiente. Serviciile oferite utilizatorilor (apelurile sistem) de către sistemul de operare constau în punerea unor parametri în registre sau stivă şi apoi execuţia unei instrucţiuni speciale de derutare (devieri), numite şi *apel nucleu* sau *apel supervizor*.

Această instrucţiune face comutarea calculatorului din *modul utilizator* în *modul nucleu* (*supervizor*) şi transferă controlul sistemului de operare (figura 1.4).

În prima fază (*1*) sistemul de operare examinează parametrii apelului pentru a determina care funcţie sistem este cerută. Apoi sistemul indexează o tabelă dispecer (*2*) cu valoarea K a apelului cerut. De aici se obţine adresa rutinei de serviciu (*3*), care va efectua funcţia apelată. În final sistemul de operare redă controlul (*4*) programului utilizator.

Structura de bază a acestui tip de sisteme de operare cuprinde:

a) un program principal care apelează procedura de serviciu cerută;

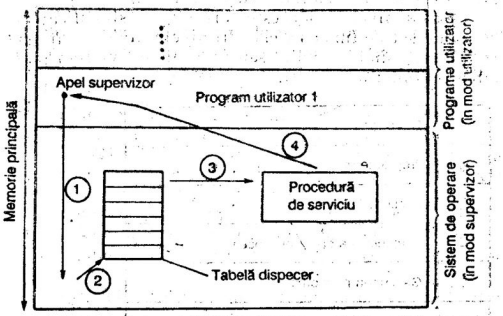


Fig. 1.4. Apelul unei funcţii sistem.

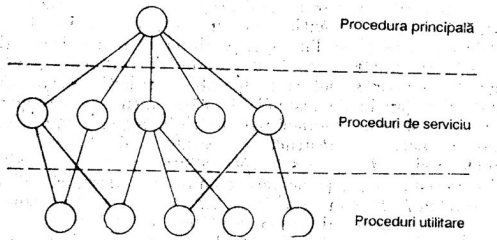


Fig. 1.5. Structura sistemului de operare monolitic.

b) o mulțime de proceduri de serviciu care tratează apelurile sistem;

c) o mulțime de proceduri utilitare care ajută procedurile de serviciu.

Putem deci concluziona că procedurile corespunzătoare sistemului de operare se împart în 3 niveluri (figura 1.5)

2) *Sisteme ierarhizate.* O generalizare a modelului de structură din figura 1.5 o reprezintă organizarea sistemului de operare ca o *ierarhie de niveluri*, fiecare din acestea construite pe un altul aflat dedesubtul său.

Fiecare nivel i al acestei arhitecturi stratificate furnizează un set de funcții (servicii) nivelului adiacent superior $(i+1)$, folosind la rîndul său funcțiile (serviciile) nivelului adiacent inferior $(i-1)$.

5	Operator
4	Programe utilizator
3	Gestiune I/E
2	Comunicarea operator-proces
1	Gestiunea memoriei
0	Multiprogramare și alocare procesor

Fig. 1.6. Structura ierarhică a sistemului THE.

Un exemplu în acest sens îl constituie sistemul de operare THE realizat în Olanda de E.W. Dijkstra la Politehnica din Eindhoven. Sistemul are 6 nivele (fig. 1.6). Nivelul *0* este destinat multiprogramării și alocării procesorului. El face comutarea între procese la secvențele de întreruperi externe sau de expirare a unor cuante de timp. Deasupra nivelului *0* sistemul constă din procese secvențiale, fiecare fiind programat fără a ține cont de execuția lor simultană pe același procesor. Cu alte cuvinte, nivelul *0* furnizează baza de multiprogramare a unității centrale.

Nivelul *1* gestionează memoria, alocînd spațiu pentru procese atît în memoria internă, cît și pe disc, pentru acele pagini care nu mai încap în memoria internă. Deasupra nivelului *1* procesele nu au cunoștință asupra alocării memoriei interne sau a spațiului disc la evacuare. Nivelul *1* va încărca în memoria internă acele pagini ale proceselor care sînt necesare.

Nivelul *2* gestionează comunicațiile între fiecare proces și consola operator. Deasupra acestui nivel fiecare proces are propria sa consolă operator.

Nivelul *3* gestionează perifericele de I/E, precum și bufferele de transfer al informațiilor cu acestea. Deasupra nivelului *3* fiecare proces lucrează cu periferice abstracte cu operații simple de I/E.

Nivelul *4* conține programele utilizator care nu trebuie să știe nimic despre procese, gestiunea memoriei, console și I/E. Procesul corespunzător operatorului se află localizat pe nivelul *5*.

O abordare superioară a acestui concept de sistem ierarhizat a fost făcută în sistemul de operare MULTICS. În loc de niveluri, MULTICS a fost organizat ca un ansamblu de inele concentrice, cele interioare mai privilegiate decît cele exterioare. Cînd o procedură de pe un nivel exterior dorește să apeleze pe o alta de pe un inel interior, prima execută instrucțiune TRAP, echivalentă unui apel sistem, ai cărei parametri sînt verificați și validați, înainte de a permite apelul.

În acest fel întregul sistem de operare este parte din spațiul de adrese al fiecărui proces; mecanisme hardware fac posibilă desemnarea procedurilor care să fie protejate

împotriva citirii, scrierii sau execuției. Mecanismul de protecție implementat la MULTICS este unul din cele mai bine puse la punct. Avantajul mecanismului bazat pe inele este acela că poate fi ușor extins în cadrul unor subsisteme utilizator. De exemplu, un profesor dorește să scrie un program lansat în nivelul *n*, pentru testarea unor programe ale studenților. Aceste programe se execută în nivelul n + 1, astfel încît studenții nu pot accesa, în vederea modificării notelor lor, programul profesorului aflat pe nivelul *n*.

3) *Mașini virtuale*. Pentru a ilustra acest concept să prezentăm un scurt exemplu. Sistemul de operare OS/360 de pe IBM a fost inițial un sistem *batch*. Dar mulți utilizatori doreau un sistem în *time-sharring*. Ca urmare, IBM a realizat sistemul TSS/360, care însă, după ce a consumat 50 de milioane de dolari, s-a dovedit a fi prea lent. Totuși un grup de utilizatori ai IBM de la Cambridge au realizat un sistem mult mai performant, care este acum larg folosit: VM/370 (vezi figura 1.7).

Sistemul VM/370 se bazează pe următoarea observație: sistemele în *time-sharring* furnizează (*1*) multiprogramarea și (*2*) o mașină extinsă cu o interfață mai convenabilă cu hardware-ul. Esența lui VM/370 este de a separa complet aceste 2 funcții.

„Inima" sistemului este *monitorul de mașină virtuală* care se execută pe hard-ul calculatorului și furnizează multiprogramarea mai multor mașini virtuale existente pe nivelele superioare. Totuși, spre deosebire de alte sisteme de operare, mașinile virtuale nu sînt și mașini extinse, cu fișiere și alte facilități. Ele sînt *exact* copii ale hardware-ului, incluzînd modul de lucru nucleu/utilizator, I/E, întreruperi, etc., pe care le are mașina reală.

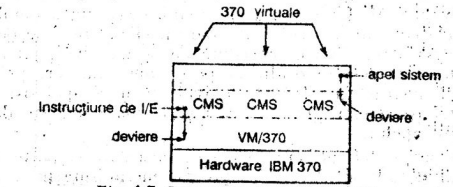


Fig. 1.7. Structura sistemului VM/370.

Deoarece fiecare mașină virtuală este identică cu hard-ul, fiecare poate executa orice sistem de operare realizat pentru această mașină; ca urmare diferitele mașini virtuale pot executa sisteme de operare diferite: OS/360 sau CMS — un sistem de operare interactiv pentru lucru multi-user.

4) *Modele „client-server"*. Sistemul VM/370 a cîștigat în simplitate și flexibilitate prin mutarea unei importante părți a sistemului de operare tradițional (implementat în mașina extinsă) într-un nivel superior, CMS. Totuși VM/370 este destul de complex ca program, deoarece simularea unui număr de mașini de tip 370 nu este foarte simplă.

O tendință modernă în sisteme de operare este de a dezvolta această idee prin mutarea de cod și mai mult în nivelele superioare, în afara unui *nucleu minimal*. Modul obișnuit este de a implementa cele mai importante funcții ale sistemului de operare în procesele utilizator. Cererea unui serviciu, cum ar fi citirea unui bloc de fișier, de către un proces utilizator (numit acum *proces client*) va conduce la emiterea unei cereri la un proces server (executant), care la sfîrșit va emite un răspuns.

În modelul din figura 1.8 nucleul va trebui să gestioneze comunicațiile între *clienți* și *executanți*. Pentru a diviza sistemul de operare în părți funcționale, fiecare din executanți gestionează anumite funcții ale sistemului, cum ar fi fișierele, procesele, memoria etc. În plus, deoarece executanții se execută ca procese în modul utilizator și nu în mod nucleu, ele nu au acces direct la hardware.

Un alt avantaj al acestui model este adaptabilitatea sa la sistemele distribuite (fig. 1.9). Dacă un client comunică cu un executant prin trimiterea unui mesaj, clientul nu trebuie să știe dacă mesajul este preluat local sau la distanță, pe un alt calculator, printr-o rețea de teleprelucrare. Din punct de vedere client lucrurile sînt similare: trimite mesajul și obține răspuns.

Schemele prezentate au sugerat faptul că nucleul are ca unică funcție gestiunea transmiterii de mesaje. Dar și alte funcții, cum ar fi gestiunea registrelor de I/E, sînt dificil de executat în spațiul utilizator. Pentru rezolvarea acestei probleme există 2 soluții. Una constă în existența cîtorva procese executant (cum ar fi driverele de I/E) care să se

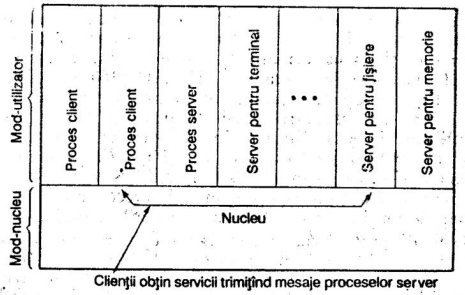


Fig. 1.8. Modelul client — executant.

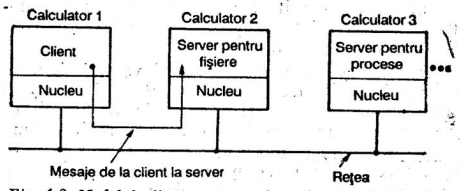


Fig. 1.9. Modelul client — server în sistemele distribuite

execute în modul nucleu, cu acces complet la hardware, dar care să comunice cu celelalte procese în acelaşi mod, prin mesaje.

O altă posibilitate este de a implementa în nucleu un mecanism minimal, care să recunoască, de exemplu, că anumite mesaje, către o anumită adresă, trebuie luate de nucleu, urmînd să fie lansate operaţiile de I/E cu perifericul respectiv. Deciziile însă şi gestiunea se fac în continuare la nivelul proceselor-executanţi.

1.5. GESTIUNEA PROCESELOR

1.5.1. Noţiunea de proces

Procesul, un concept-cheie al sistemului de operare, reprezintă o abstractizare a unui program în execuţie. Un calculator satisface mai multe sarcini simultan; cum ar fi lucrul în unitatea centrală şi lucrul cu perifericele. Ca urmare, unitatea centrală satisface simultan mai multe programe, într-un *pseudoparalelism*; acesta constă într-o comutare rapidă a unităţii centrale între diferite programe, atribuindu-se fiecăruia cîteva fracţiuni de timp. Sistemul de operare este cel care trebuie să ţină evidenţa tuturor acestor activităţi paralele, asigurînd astfel distribuţia uniformă a resurselor între procesele care se execută simultan.

Tot ce se execută pe un calculator, inclusiv sistemul de operare, este organizat ca o suită de *procese secvenţiale* sau pe scurt *procese*, care reprezintă tocmai programe în execuţie, inclusiv contextul acestora (date, registre generale, contorul programului — PC, pointerul de stivă — SP).

Conceptual, fiecare proces are propria sa unitate centrală virtuală; în realitate însă în sistemele monoprocesor, unitatea centrală se comută de la un proces la altul, executîndu-se aşa-numita *multiprogramare*.

În figura 1.10 se indică modul cum sînt multiprogramate 3 programe prezente simultan în memorie (*a*). Ele dau naştere la 3 procese (*b*), fiecare cu propriul său contor program (*PC*) şi fiecare executat în mod independent de celelalte. În (*c*) se indică modul cum sînt executate în timp cele 3 procese, cîte unul singur la un moment dat.

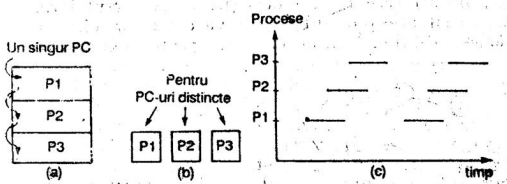


Fig. 1.10. Multiprogramarea a trei procese.

Afectarea (comutarea) unității centrale la procese se face în funcție de cererile lor de I/E sau după epuizarea unor cuante de timp, astfel încît sistemul de operare să activeze cît mai uniform toate procesele.

Se observă că diferențele dintre program și proces este subtilă, (dar esențială). Să reținem că un proces conține următoarele componente: programul, datele de intrare, datele de ieșire și starea (contextul).

În sistemele de operare simple, care execută o singură aplicație la un moment dat, este posibil să se dispună de toate procesele necesare executării, încărcate la inițializarea sistemului de operare. În sistemele mai complexe, există facilități de creare și distrugere a proceselor. În UNIX, de exemplu, directiva *fork* creează o copie identică cu programul apelant, iar directiva *exec* va introduce în această copie codul unui alt program dorit. Prin acest mecanism, *fork-exec*, se pot crea *arbori de procese*, în care fiecare proces poate avea un „tată", și mai mulți „fii".

Deși un proces este o entitate separată, adesea este necesar să interacționăm cu alte procese.

De exemplu, în suita de comenzi UNIX:

cat f1 f2 | grep student

primul proces încărcat, *cat*, va concatena 2 fișiere, *f1* și *f2*, iar următorul proces, *grep*, selectează toate liniile care conțin cuvîntul „student".

Cele 2 procese, *cat* și *grep*, se lansează în același timp; dar în funcție de viteza relativă a celor 2 procese se poate ca *grep* să fie gata de lansare dar să nu aibă intrările pregătite, intrări care reprezintă ieșiri ale procesului *cat*. În această situație procesul *grep* se va *bloca* pînă cînd are intrările pregătite. Însă un proces se poate bloca și datorită

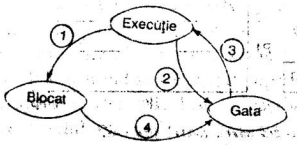


Fig. 1.11. Diagrama de stări ale unui proces.

faptului că sistemul de operare alocă unitatea centrală unui alt proces prioritar. În figura 1.11 se indică *diagrama de stări* ale unui proces.

Se observă că sînt posibile 3 stări:

1) *în execuție* — pentru procesul care ocupă unitatea centrală;
2) *blocat* — pentru procesele care nu se pot executa pînă ce nu se produc anumite evenimente externe (sfîrșit de operație de I/E, recepție mesaj de la alt proces etc.);
3) *gata* — pentru procesele care pot fi executate dar au fost stopate de procesul în execuție.

Diagrama indică cele 4 tranziții posibile:

1) Tranziția apare atunci cînd un proces nu poate continua execuția sa; de obicei face o operație de intrare/ieșire;

2) Tranziția apare cînd un proces, ocupînd prea mult unitatea centrală, este oprit după epuizarea unei cuante de timp, planificatorul alocînd unitatea centrală unui alt proces;

3) Tranziția apare cînd un proces, prioritar dintre cele din starea *gata*, primește drept de execuție în unitatea centrală;

4) Tranziția are loc cînd apare un eveniment extern așteptat de un proces. În acest moment procesul intră în starea *gata*, fiind capabil să se execute în unitatea centrală.

Sistemul de operare are implementați algoritmi care să permită *planificatorului de procese* să decidă căror procese le alocă unitatea centrală.

Utilizînd acest *model cu procese*, devine mai ușor să explicăm ce se întîmplă în sistemul de operare. O serie de procese încarcă programe care execută comenzile tastate de utilizator. Alte procese sînt parte din sistem și gestionează task-uri, cum ar fi gestiunea fișierelor, gestiunea driverelor de intrare/ieșire etc. De exemplu, cînd apare o întrerupere de la disc, sistemul de operare decide să oprească procesul curent și să încarce procesul disc, care este blocat în așteptare de întrerupere. Acesta tratează întreruperea și blocul disc, fiind citit; procesul ce aștepta poate fi deblocat și devine eligibil.

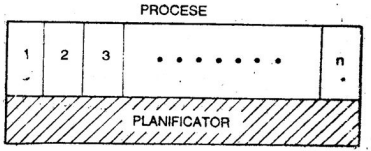


Fig. 1.12. Modelul bazat pe procese.

Această viziune asupra sistemului de operare dă naștere modelului prezentat în figura 1.12. Pe cel mai de jos nivel este *planificatorul* care tratează întreruperile, face oprirea și startarea proceselor. Restul sistemului de operare este structurat în procese secvențiale. Planificatorul asigură, pe lîngă gestiunea întreruperilor, și comunicațiile între procese.

Pentru a *implementa* acest model cu procese, sistemul de operare menține o tabelă sau matrice de structuri, numite *tabela proceselor* (TP), cu cîte o intrare pentru fiecare proces. Aceste intrări conțin informații despre starea proceselor, PC-ul, SP-ul, alocarea memoriei, starea fișierelor deschise, informații de planificare și alte date care trebuie activate atunci cînd procesul trece din starea „execuție" în starea „gata".

Să vedem acum, pe baza unui exemplu din sistemul UNIX, cum se creează imaginea virtuală de execuție a mai multor procese pe un calculator cu o singură unitate centrală și mai multe periferice. Sistemul de operare asociază fiecărei clase de periferice cîte o locație la începutul memoriei, numită *vector de întrerupere*; acesta conține adresa procedurii de tratare a întreruperii. Să presupunem că procesul *i* este în execuție cînd apare întreruperea de la disc. PC-ul, SP-ul și cîteva din registrele generale ale acestui proces sînt puse în stiva de întrerupere hardware; apoi calculatorul sare la adresa specificată în vectorul de întreruperc al discului. Acestea sînt singurele acțiuni ale hard-ului; în continuare acționează soft-ul.

Procedura de tratare a întreruperii începe prin a salva registrele generale în TP, pe linia *i* a procesului întrerupt. Se va lua apoi informația depusă în stivă și se va copia în TP. Aceste acțiuni sînt realizate de o mică procedură în limbaj de ansamblare.

Interconectarea între procese se poate face prin mesaje. Ca urmare, în pasul următor se construiește un mesaj pentru a fi emis la procesul disc care este blocat în așteptare de mesaje. Mesajul îi spune că a apărut întreruperea, pentru a-l distinge de alte mesaje venite de la procesele utilizator și care îi cer să citească sau să scrie blocuri disc. Ca urmare, starea procesului disc trece din „blocat" în „gata" și se apelează planificatorul. În UNIX procesele au priorități pe baza cărora se alege procesul ce intră în execuție.

Procedura în limbaj C apelată de procedura de tratare a întreruperii (în limbaj de ansamblare) returnează controlul primeia care va încărca registrele pentru un nou proces curent și îl va starta.

1.5.2. Cooperarea proceselor

Procesele, în cursul existenței lor, comunică între ele. De exemplu, cînd un proces utilizator dorește să citească dintr-un fișier, trebuie să-i „spună" ce vrea unui proces fișier. Acesta, la rîndul său, va trebui să-i „spună" unui proces disc să citească blocul dorit. Realizarea acestor dorințe antrenează și un proces de confirmări, prin transmiterea unor informații în sens invers. În cadrul acestui paragraf vom analiza problemele complexe care apar în procesul de comunicație și cooperare între procese.

1.5.2.1. *Condițiile de concurență*

În cadrul sistemelor de operare procesele utilizează în comun o zonă de memorie (internă sau disc), în care fiecare poate scrie sau citi. Pentru a vedea cum comunică între ele procesele să considerăm exemplul unui *program spooler* care tipărește fișiere text. Cînd un proces dorește să listeze un fișier, el va introduce numele acestuia în *directorul* spooler-ului. Un alt *proces de tipărire*, periodic, verifică dacă există fișier de tipărit și după ce îl listează șterge liniile corespunzătoare din director.

Să presupunem că intrările în director sînt numerotate cu 1, 2, ..., fiecare din ele avînd posibilitatea să rețină numele și datele unui fișier de tipărit. După cum se vede în figura 1.13, există 2 variabile care sînt utilizate în comun: *out* — care pointează următorul fișier ce trebuie

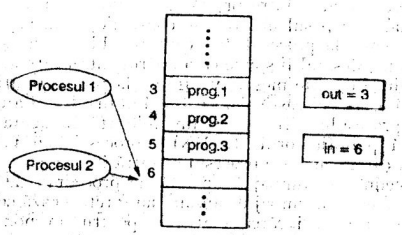


Fig. 1.13. Concurența proceselor 1 și 2.

listat și *in* — care pointează următoarea intrare liberă în director.

Aceste 2 variabile trebuie accesate de toate procesele, deci vor fi folosite în comun.

În situația concretă din figura 1.13, procesele 1 și 2 decid simultan să pună cîte un fișier de listat în directorul spooler-ului. În acest caz se poate ajunge la următoarea situație nedorită: procesul 1 citește *in* și memorează valoarea 6 într-o variabilă locală. În acest moment apare o întrerupere și planificatorul dă comanda procesului 2. Acesta citește și el variabila *in*, memorează aceeași valoare 6, ocupă linia 6 din director și actualizează valoarea lui *in* la 7. Cînd se va relua procesul 2, el va ocupa tot linia 7 din director și va incrementa *in* cu 1 la valoarea 9; ca urmare, fișierul procesului 2 nu va mai fi listat. Astfel de situații, cînd mai multe procese folosesc în comun aceeași resursă, sînt numite *condiții de concurență*, și ele trebuie stăpînite de sistemul de operare.

1.5.2.2. *Secțiuni critice*

Soluția pentru evitarea condițiilor de concurență este de a găsi o cale care să împiedice ca mai multe procese să acceadă în același timp la variabilele (fișierele) comune.

Cu alte cuvinte, se dorește o *excluziune mutuală* a proceselor cînd utilizează resurse comune.

Acele părți din programe în care este posibil accesul la date comune se numesc *secțiuni critice*. Pentru ca 2 procese să coopereze în condiții bune, nu este suficient să se elimine condițiile de concurență, ci trebuie îndeplinite 4 condiții, adevărate reguli de „comportament" în programarea proceselor:

1) Să nu existe 2 procese care să execute simultan secțiunile lor critice;

2) Nu trebuie făcute presupuneri în ceea ce privește viteza proceselor sau numărul de unități centrale;

3) Nici un proces, oprit în afara secțiunii critice, nu trebuie să blocheze alte procese.

4) Nici un proces nu trebuie să aștepte oricît de mult pentru a intra în secțiunea sa critică.

În continuare se vor analiza mai multe soluții pentru rezolvarea problemei cardinale a excluziunii mutuale.

1.5.2.3. *Excluziunea mutuală prin așteptare-ocupare („busy-waiting")*

Vom prezenta cîteva soluții pentru realizarea excluziuii mutuale, care permite ca în timp ce un proces este ocupat cu actualizarea unei date din regiunea sa critică, nici un alt proces să nu intre în regiunea critică.

1) *Dezactivarea întreruperilor* reprezintă cea mai simplă soluție. La intrarea în regiunea critică procesul dezactivează toate întreruperile, reactivîndu-le la ieșire. Ca urmare, nu poate să apară nici o întrerupere care să oprească procesul pe parcursul actualizării datelor comune. Soluția încă prezintă un inconvenient major: dă putere proceselor să dezactiveze sistemul de întreruperi. Dacă un proces, dintr-o eroare de programare, nu le mai activează la ieșire din regiunea critică, întregul sistem de operare „moare".

2) *Variabile de blocare*. Să considerăm că există o singură variabilă de blocare, inițial pe 0. Cînd un proces intră în regiunea critică, testează variabila; dacă o găsește pe 0 (regiune liberă) o va pune pe 1 și va intra în regiunea critică. Dacă variabila are valoarea 1, procesul va aștepta

punerea ei, de către alte procese, pe 0. Din nefericire ideea conține același inconvenient observat și în exemplul *spooler*ului.

3) Stricta alternanță. Soluția propusă presupune următorul algoritm:

```
cît timp (TRUE) execută
    cît timp (var < > 0)
        execută /*așteaptă*/          (a)
    * secțiune — critică
      sem : =1
    * secțiune — necritică

cît timp (TRUE) execută
    cît timp (var < > 1)
        execută /* așteaptă */        (b)
    * secțiune — critică
      sem : = 0
    * secțiune necritică
```

Este vorba de 2 procese, (*a*) și (*b*), care concură în utilizarea unei variabile comune, actualizată în regiunile critice. Variabila *sem*, inițial pe 0, permite la început procesului (*a*) să intre în regiunea sa critică. Procesul (*b*) testează și așteaptă pînă ce *sem* se va pune pe 1, la sfîrșitul regiunii critice a lui (*a*). Testarea continuă a unei variabile pînă cînd primește o valoare dorită se numește *așteptare ocupată* („*busy-waiting*").

Soluția propusă are însă unele inconveniente. Să presupunem că (*b*) a ieșit din regiunea critică, á pus *sem* pe 0 și a intrat în regiunea necritică. În acest moment, (*a*) — ieșit și el din regiunea critică — îl pune pe *sem* pe 0 și intră în zona necritică. Procesul (*a*), dacă revine la început și încearcă intrarea în regiunea critică, va fi blocat deoarece *sem* este pe 1, iar (*b*) — lucrînd mai mult în regiunea necritică — nu va debloca variabila. În acest fel se violează

condiția (*3*) din § 1.5.2.2; procesul (*a*) este blocat de un proces (*b*) care nu lucrează în regiunea critică. Soluția este indicată doar în cazul unor procese similare (identice).

4) Soluția Peterson. Peterson combină ideea variabilei de blocare cu cea a variabilei de „comutare" din soluția de strictă alternanță, publicînd următorul algoritm, bazat pe 2 proceduri *intrare-regiune* și *ieșire-regiune*:

```
procedura intrare-regiune (proces) este /* apelez cu numărul
                                          0 sau 1 al procesului */
    alt-proces:=1—proces      /* numărul celuilalt proces */
    interesat [proces] : = true /* marchez că sînt interesat */
    sem: = proces             /* se setează fanionul */
    cît timp (sem = proces și interesat [alt-proces] = true)
        execută               /* instrucțiune nulă */
    sfîrșit

procedura ieșire-regiune (proces) este
    interesat [proces] : = false  /* se indică părăsirea
                                     regiunii critice */
sfîrșit
```

Înainte de intrarea în regiunea critică fiecare proces apelează procedura *intrare-regiune* cu parametru numărul său de proces (0 sau 1). Această procedură îl face eventual să aștepte pînă la eliberarea regiunii critice. După terminarea actualizării variabilelor comune, se apelează procedura *ieșire-regiune* care va permite altor procese să ocupe regiunea. Se folosește un vector *interesat* care are 2 elemente corespunzătoare celor 2 procese; valorile lor pot fi *true* (cînd procesul dorește ocuparea) și *false* (cînd procesul nu dorește să ocupe regiunea). Să presupunem cazul cel mai nefavorabil cînd cele 2 procese apelează simultan procedura *intrare-regiune*, ambele punînd valoarea numărul de proces în variabila *sem*. Dacă procesul 1 pune ultima valoare, *sem* va avea valoarea 1. Cînd ambele procese intră în ciclul *cît timp*, procesul 0 nu execută ciclul deoarece găsește condiția adevărată și ocupă regiunea. Procesul 1 însă se va bloca și va aștepta eliberarea regiunii.

1.5.2.4. *Excluziunea — mutuală prin așteptare — activare („sleep and wakeup")*

Ultima soluție pentru excluziune mutuală, prezentată de Peterson, este corectă, dar are defectul că implică o așteptare-ocupată; ca urmare, un proces care dorește să intre într-o regiune critică ce este ocupată se va plasa într-o buclă, ciclînd pînă la eliberarea resursei. Acest lucru nu numai că ocupă inutil unitatea centrală, dar poate conduce și la blocarea reciprocă a 2 procese, dacă acestea au priorități diferite (dacă procesul cu prioritate ridicată ocupă unitatea centrală în bucla de așteptare, iar cel cu prioritate scăzută ține resursa ocupată neputînd să o elibereze pînă ce obține unitatea centrală, blocajul este reciproc).

Soluția care se impune constă în existența unor primitive prin care un proces să se blocheze pînă la eliberarea resursei. Se poate folosi o pereche de primitive (apeluri sistem), numite generic *activare* și *așteptare*. *Așteptarea* este un apel sistem care permite operatorului să se suspende (în starea blocare) pînă cînd un alt proces îl „trezește". Primitiva *activare* are ca parametru procesul ce va fi „trezit", trecîndu-l în starea „gata".

Vom analiza modul de utilizare a celor 2 primitive într-un exemplu cu cunoscuta problemă *producător-consumator* (sau a bufferului limitat).

Două procese utilizează în comun un ansamblu de buffere de dimensiune fixă. Unul din ele — producătorul — scrie informații în buffere, celălalt — consumatorul — scoate informații din buffere.

Dificultatea apare atunci cînd producătorul dorește să scrie, dar setul de buffere este plin. Soluția este să aștepte (primitiva *așteptare*) pînă cînd consumatorul creează cel puțin un buffer liber. Similar se pune problema dacă consumatorul dorește să citească din setul de buffere, iar acestea sînt toate goale (nu există informație depusă de producător).

Problema pare simplă, însă apare același tip de conflict ca în problema spooler-ului. Pentru a reține numărul de buffere completate din set se folosește variabila *contor*, care la început este 0 și poate crește pînă la n (numărul de buffere din set). Ca urmare, producătorul se pune în așteptare cînd *contor* este n (nu mai sînt buffere libere), iar consumatorul așteaptă cînd *contor* este 0 (nu mai sînt buffere complete). Vom prezenta în continuare algoritmul procedurilor *producător* și *consumator*:

```
procedura producător este                          X-ON ; X-OFF.
    cît timp (true) execută      /* ciclu infinit */
        * produce-buffer
        dacă (contor = n) atunci așteptare ( )
                        /* dacă setul de buffere e plin — așteptare */
        * introducere-buffere    /* pune buffer în set */
          contor: = contor + 1 /* incrementează contor */
        dacă (contor = 1) atunci
                    activare (consumator) /* s-a completat
                                             primul buffer */
sfîrșit

procedura consumator este
    cît timp (true) execută                  /* ciclu infinit */
        dacă (contor=0) atunci așteptare ( ) /* nu există
                                                buffere complete */
        * scoate-buffer
          contor: = contor — 1
        dacă (contor = n — 1) atunci
                    activare (producător) /*nu au fost
                                            buffere libere */
        * consumă-buffere
sfîrșit
```

Condițiile de concurență apar datorită accesului la resursa comună *contor*. Setul de buffere este gol și consumatorul tocmai a citit pe *contor* pentru a vedea, dacă este 0. În acest moment planificatorul hotărăște să stopeze pro-

cesul consumator şi startează procesul producător. Acesta introduce în set un buffer completat, incrementează *contor* care devine 1; în acest moment producătorul execută *activare* pentru a „trezi" consumatorul. Dar acesta nu este în aşteptare; cînd se activează, el crede că variabila *contor* este 0 şi se pune în aşteptare. Cînd şi producătorul se pune în aşteptare cele 2 procese se blochează definitiv.

1.5.2.5 *Excluziunea mutuală prin semafoare*

E.W. Dijkstra propune o soluţie superioară, constînd în utilizarea unei variabile întregi — numită *semafor* — care să contorizeze numărul de activări salvate pentru viitoarea utilizare, şi nu doar un bit ca la metoda anterioară. Un semafor are valoare 0 cînd indică faptul că nu s-au salvat activări sau o valoare pozitivă — n — cînd sînt în aşteptare n activări.

Se defineşte un *semafor*, S, ansamblul format din variabilele întregi $P(s)$ şi firul de aşteptare asociat, $f(s)$. Politica de gestiune a firului de aşteptare poate fi variabilă de la o implementare la alta. La crearea lui $e(s)$, $f(s)$ este vid.

Dijkstra propune 2 primitive P şi V, pentru a acţiona asupra semafoarelor: generalizarea lui *activare* şi *aşteptare* din soluţia anterioară.

(a)

```
procedura P(s) este
    e(s) := e(s) − 1
    dacă e(s) <0 atunci
        stare(r) := blocat
        * pune r în firul f(s)
sfîrşit
```

Această primitivă este executată de un proces oarecare n, atunci cînd doreşte ocuparea unei resurse. Se verifică dacă semaforul a devenit negativ după decrementare. În caz afirmativ nu mai există resurse şi procesul n este pus în firul de aşteptare al semaforului S. Se observă că dacă $e(s)>0$, valoarea $e(s)$ indică numărul de resurse (activări)

disponibile. În caz că $e(s)<0$, valoarea $e(s)$ indică numărul de cereri de resursă nesatisfăcute şi aflate în firul de aşteptare.

(b)

```
procedura V(s) este
    e(s) := e(s) + 1
    dacă e(s)<=0 atunci
        * scoate un proces q din firul f(s)
        stare (q) := activ
sfîrşit
```

Aceste primitive se apelează de un proces care eliberează o resursă. Procesul incrementează semaforul şi dacă valoarea lui este negativă sau zero, ceea ce înseamnă că există procese în fire de aşteptare, scoate unul şi îl activează.

Se pot enumera cîteva proprietăţi ale semafoarelor:

(1) Un semafor nu poate fi iniţializat la valori negative, dar poate deveni negativ după un număr de operaţii, P.

(2) Dacă notăm cu:

$nP(s)$ — numărul de operaţii P asupra lui S,
$nV(s)$ — numărul de operaţii V asupra lui S,
$e_0(s)$ — valoarea iniţială a lui S,
există relaţia:

$$e(s) = e_0(s) - nP(s) + nV(s).$$

(3) Fie $nf(s)$ numărul de procese care după ce au făcut $P(s)$ fie că nu au fost blocate, fie că au fost blocate, dar apoi deblocate cu $V(s)$. Avem relaţia:

$$nf(s) <= nP(s).$$

(4) Efectul primitivelor P şi V asupra lui $nf(s)$ este următorul:

```
** efectul P(s) este
    nP(s):=nP(s)+1
    dacă nP(s)<=e0(s)+nV(s)
        atunci nf(s): =nf(s)+1
sfîrşit
```

```
** efectul V(s) este
        nV(s) : =nV(s)+1
        dacă nP(s)=>e0(s)+nV(s)
            atunci nf(s):=nf(s)+1
sfîrşit
```

(5) Execuţia primitivelor P şi V lasă invariantă relaţia: $nf(s)=\min(nP(s),\ e0(s)+nV(s))$.

Această relaţie, aparent complicată, se poate interpreta simplu: asimilînd semaforul cu o barieră, operaţia P reprezintă o cerere de trecere. Valoarea $e0(s)+nV(s)$ reprezintă numărul total de autorizaţii de trecere disponibile la un moment dat. Relaţia de mai sus exprimă că numărul efectiv de treceri peste barieră este cel mai mic dintre numărul de cereri şi numărul de autorizaţii.

Se va prezenta în continuare aceeaşi problemă *producător-consumator* („reader-writer"), în care este vorba de un set de buffere. Procesele *producător* completează cîte un buffer cu informaţii, în timp ce procesul *consumator* foloseşte aceste informaţii golind bufferele. Pentru implementarea acestor 2 proceduri se folosesc 3 semafoare:

plin — pentru numărul de buffere din set ocupate (iniţial *plin* = 0)

gol — pentru numărul de buffere din set libere (iniţial *gol* = N)

mutex — pentru protejarea accesului în setul de buffere (iniţial *mutex* = 1); este un semafor liber.

```
procedura producător ( ) este
    cît timp (true)              /* ciclu infinit */
        * produce-buffer (buf)   /* generează informaţii
                                    pentru scriere în buffer */
        P(gol)          /* cer acces în setul de buffere */
        P(mutex)        /* intrare în regiunea critică */
        * scriere-buffer (buf) /* scrie informaţia în buffer */
        V(mutex)        /* părăsirea regiunii critice */
        V(plin)         /* activez un proces în aşteptare */
sfîrşit
```



```
procedura consumator ( ) este
    cît timp (true)              /* ciclu infinit */
        P(plin)         /* decremetează contorul plin */
        P(mutex)        /* intrare în regiunea critică */
        * scoate-buffer (buf)    /* iau buffer din set */
        V(mutex)        /* părăsirea regiunii critice */
        V(gol)          /* incrementează contorul gol */
        * consumă-buffer (buf)   /* folosesc informaţia */
sfîrşit
```

Pe lîngă aspectele pozitive ale soluţiei, putem semnala şi unele limite ale metodei: în sistemele mari, complexe, devine dificilă folosirea semafoarelor; există posibilitatea ca din cauza unor erori de programare să se sară în mod accidental peste executarea lui P, ceea ce duce la accesul neprotejat la regiunile critice; în mod analog, dacă se sare peste execuţia lui V se poate ajunge la fenomenul de blocare („deadlock"); nu se poate verifica prin procesul de compilare utilizarea corectă a semafoarelor.

1.5.2.6 *Contoare eveniment*

Reed şi Kanodia propun o soluţie la problema producător-consumator, fără a fi nevoie de excluziunea mutuală. În acest scop se utilizează un tip special de variabile, numite *contor-eveniment*.

Se definesc 3 operaţii asupra unui contor-eveniment, e:

1) *citire*(e) — returnează valoarea curentă a lui e
2) *avans*(e) — se incrementează e cu 1
3) aştepta(e,w) — se aşteaptă „cît timp" e are o valoare mai mică ca w

În implementarea problemei producător-consumator nu se utilizează funcţia *citire*, necesară în alte probleme de sincronizare. Se vor folosi două contoare-eveniment: *in* care marchează numărul de buffere inserate în set şi *out* care memorează numărul de buffere scoase din set.

```
procedura producător ( ) este
    secv:=0
    cît timp (true) execută
        * produce-buffer (buf)          /* produc informație */
        secv:=secv+1   /* contorizează bufferele produse */
        așteptare (out, secv−n) /* aștept pînă ce este loc în set*/
        * scrie-buffer (art)        /* scriu informație în buffer */
        avans (in)          /* informează consumatorul despre
                                        bufferul produs */
sfîrșit
```

```
produce consumator ( ) este
    secv: =0
    cît timp (true) execută
        secv:=secv+1          /* numărul de buffere scoase din
                                        set */
        așteptare (in, secv) /* aștept pînă există buffere */
        * scoate-buffer (buf)       /* scoate buffer din set */
        avans (out) /* producătorul este informat că un buffer
                                    a fost eliberat */
        * consumă-buffer (buf)
sfîrșit
```

După ce producătorul a creat un nou buffer, verifică prin așteptare, dacă este loc în set pentru a-l introduce. Cum inițial out =0 și (secv-n) este negativ (*n* este numărul de buffere din set), producătorul nu se va mai bloca. În cazul în care producătorul se blochează, consumatorul îl va debloca cînd va face contorul-eveniment 1. Logica consumatorului este simplă: înainte de a încerca să scoată cel de-al *k*-lea buffer din set, așteaptă pînă ce *in* devine *k*, adică pînă ce producătorul a pus cel puțin *k* buffere în set.

1.5.2.7 *Sincronizarea prin variabile comune*

Hoare și B. Hansen au propus un mecanism de sincronizare a proceselor bazate pe conceptul de *variabilă comună*.

O variabilă este numită comună dacă 2 sau mai multe procese îi pot modifica valoarea. Declararea ei se face, într-o notație asemănătoare limbajului PASCAL, astfel:

var v: *sharred integer*

sau

var v: *sharred record*

b: *real*

c: *boolean*

end;

Prin definiție, o variabilă comună este accesibilă numai într-o regiune critică, iar regiunile critice referitoare la aceeași variabilă comună se exclud reciproc. O *regiune critică* se definește astfel:

region v *do*

begin

end;

Secvența de instrucțiuni cuprinsă între *begin* și *end* se execută ca și cînd ar fi încadrate de 2 operații *P* și *V* asupra unui semafor binar. Avantajul față de primitivele lui Dijkstra este că în cazul acestei soluții se pot face verificări în procesul de compilare asupra utilizării corecte a variabilelor comune, numai în regiunile critice.

Conceptele de *regiune critică* și *variabilă comună* permit o descriere simplă a excluziunii mutuale, dar nu și a coope-

rării. În vederea satisfacerii acestei cerințe s-a introdus funcția *await*, putîndu-se crea *regiuni critice condiționate*:

```
region v do
        begin
          .
          .
          .
          await  cond;
          .
          .
          .
          end;
```

La întîlnirea instrucțiunii *await* se verifică dacă valorile componentelor variabilei comune *v* satisfac condiția *cond*. În caz afirmativ, procesul continuă execuția; în caz contrar, procesul va fi blocat pînă ce, prin acțiunea altor procese, condiția *cond* va fi satisfăcută. Codul generat la compilare va trebui să conțină și porțiuni pentru gestiunea firului de așteptare asociat variabilei comune, format din procesele care doresc să acceadă la variabilă.

Dezavantajul implementării acestui concept în cadrul unui limbaj de programare constă în faptul că fiecare proces trebuie să conțină propria sa porțiune de cod pentru a se opera asupra variabilei comune.

Hoare și B. Hausen vor introduce un nou concept — *monitor* — care permite un nivel mai înalt de abstractizare prin definirea variabilei comune și a operațiilor permise asupra ei (prin proceduri) ca o entitate separată, utilizată de fiecare proces.

1.5.2.8. *Sincronizarea prin monitoare*

Un *monitor* definește o structură de date folosită în comun de mai multe procese, precum și toate operațiile permise asupra ei. Aceste operații sînt specificate prin *proceduri monitor*. Un monitor definește de asemenea o operație inițială care se execută în momentul creării structurii de date.

Accesul la datele monitorului se poate face numai prin procedurile acestuia, iar dintre acestea cel mult una poate

fi activată la un moment dat. Indiferent de numărul de procese care utilizează setul de date comune declarat ca monitor, există o singură copie a codului și a datelor acestuia.

Iată cum se definește în PASCAL CONCURENT structura de tip monitor:

```
type  nume = monitor (parametrii_formali);
        declarații de date comune ale monitorului
        procedure entry nume (parametrii_formali)
                  begin . . . end;
        procedure entry nume (parametrii_formali)
                  begin . . . end;
        ..........
        procedure nume (parametrii_formali)
                  begin . . . end;
        procedure nume (parametrii_formali)
                  begin . . . end;
        ..........
        begin
            secvență inițială
        end;
```

După cum se observă există 2 categorii de proceduri: *procedure entry*, care reprezintă procedurile monitor, apelate de procese în vederea accesului la date comune, și *procedura*, care constituie definițiile de proceduri locale monitorului.

Apelul unei proceduri monitor se face astfel:

nume_monitor . nume_procedura (parametrii actuali);

Monitoarele rezolvă direct doar excluziunea mutuală. Pentru problemele de cooperare (sincronizare) între procese a fost necesară introducerea unor construcții speciale: *delay* și *continue*.

Cînd o anumită condiție așteptată de un proces ce execută o procedură monitor nu este satisfcăută, procesul este introdus printr-un *delay* în firul de așteptare asociat condiției respective. După un interval de timp nedefinit, cînd execuția unui alt proces în acel monitor face ca această condiție să fie îndeplinită, în urma unei operații *continue* se activează procesul prioritar din firul de așteptare.

Regula fundamentală în utilizarea monitoarelor constă în a fi părăsite cît mai repede de către procese după folosirea structurilor de date comune. De aici derivă următoarele 2 probleme de implementare:

1) Apelurile simultane la același monitor conduc la necesitatea excluziunii mutuale, lucru realizat printr-un semafor asociat monitorului. În firul de așteptare al acestui semafor se află procese cărora nu li s-a permis încă accesul.

2) Există o serie de fire de așteptare interne, unde procesele sînt introduse cu *delay*. În aceste momente, monitorul devenind liber, este atribuit unui alt proces care așteaptă la intrare. În momentul în care un proces este reactivat prin *continuare* dintr-un fir intern, el trebuie să aibă imediat acces la monitor în detrimentul proceselor de la intrare.

Monitoarele fiind niște construcții cu un înalt nivel de abstractizare, au avantajul că utilizarea lor poate fi verificată în procesul de compilare. Limbajele concurente, ca PASCAL CONCURENT (Hansen) sau EUCLID (Holt), conțin astfel de structuri. Vom prezenta în continuare schița unui program care să rezolve aceeași problemă producător-consumator. Variabilele *plin* și *gol*, cu semnificația prezentată în celelalte paragrafe, sînt aici organizate ca fire de așteptare:

```
monitor producător-consumator;
  ..........
  procedure entry scrie ( ); /* procedura monitor */
    begin
      if contor = N then delay (plin);
        scrie-buffer;
        contor:=contor+1;
      if contor=1 then continue (gol);
    end;
  procedure entry citește ( ); /*procedura monitor */
    begin
      if contor = 0 then delay (gol);
        citește-buffer;
        contor:=contor-1;
      if contor = N-1 then continue (plin);
    end;
  begin                    /*procedura de inițializare */
    contor: = 0
  end;
end;
  ..........
```



```
procedure producător ( );
  begin
    while true do
      begin
        produce-buffer;
        producător consumator. scrie;
      end;
end;
procedura consumator ( );
  begin
    while true do
      begin
        producător-consumator. citește;
        consuma-buffer;
      end;
end;
```

1.5.2.9 *Sincronizare prin mesaje*

Transmiterea de mesaje, ca metodă de comunicație între procese, poate folosi 2 primitive (apeluri sistem):

send (destinație, mesaj) — pentru *emisie de mesaje*;
receive (sursa, mesaj) — pentru *recepția unui mesaj*.

În acest caz se impun unele cerințe noi, în special la procese de pe calculatoare diferite, conectate în rețea. Pentru a preîntîmpina pierderea unor mesaje sînt necesare *mesaje de confirmare*. Astfel, dacă emițătorul unui mesaj nu a primit confirmarea recepției de la receptor, după un anumit interval de timp va retransmite mesajul. Se poate întîmpla să piardă nu mesajul, ci confirmarea. De aceea, este necesar să se atașeze fiecărui mesaj cîte un număr de secvență. În plus este necesară și *autentificarea* mesajelor, care trebuie să ne asigure că se comunică cu adevăratul proces, și nu cu un „intrus". În acest caz sînt folosite *metode de criptografie computațională*.

Să analizăm modul cum poate fi soluționată problema *producător-consumator* nu cu zona de memorie comună, ci prin mesaje. Vom face presupunerea că toate mesajele au aceeași dimensiune, și că mesajele emise, dar încă nerecep-

ționate, sînt puse automat în buffere de către sistemul de operare. Se folosesc N mesaje. Consumatorul începe prin a emite N mesaje goale la producător. Acesta ia cîte un mesaj gol, îl completează cu date și îl trimite înapoi plin.

Dacă producătorul lucrează mai repede decît consumatorul, toate mesajele vor deveni pline și în așteptare la consumator. Producătorul va fi blocat, așteptînd returnarea unor mesaje goale. Cînd consumatorul lucrează mai repede, el se va bloca așteptînd sosirea unor mesaje pline.

Pentru transmiterea mesajelor se pot folosi niște structuri de date de un tip nou, numite *cutii poștale*, unde este reținut un anumit număr de mesaje, specificat la crearea cutiei. Ca urmare, adresele din *send* și *receive* reprezintă cutii și nu procese. Cînd un proces încearcă să emită într-o cutie plină, va fi în așteptare.

```
procedura producător ( ) este
    cît timp (true) execută
        * produce-buffer (buf) /* generează date și pune în
                                                    buffer */
        receive (consumator, mesaj) /* așteaptă mesaj gol */
        * construiește-mesaj (mesaj, buf) */ pregătesc mesaje
                                                pentru emisie */
        send (consumator, mesaj)    /* emite mesaj */
sfîrșit

procedura consumator ( ) este
    pentru i = 1, n execută
        send (producător, mesaj)  /* emite N mesaje goale */
    cît timp (true) execută
        receive (producător, mesaj)  /* se citește mesaj */
        * extrage-buffer (mesaj, buf)
        * consumă-buffer (buf)
        send (producător, mesaj)    /* emite mesaj gol */
sfîrși!
```


Dacă, în procesul de emisie, *send* se face înainte de *receive*, acesta este blocat pînă ce s-a executat *receive* la procesul receptor. În acest fel mesajele pot fi copiate direct de la emițător la receptor, fără a mai fi necesară trecerea lor prin buffer. Similar, dacă se execută mai întîi *receive* în procesul receptor, acesta este blocat pînă ce nu s-a realizat *send*. Este un concept puțin flexibil, cunoscut sub numele de *randez-vous* și implementat în limbajul ADA.

1.5.2.10 *Sincronizarea prin randez-vous*

Conform unei observații a lui Conway, sincronizarea și transmiterea de date între 2 procese sînt acte inseparabile. Două procese paralele (concurente) trebuie să se sincronizeze pentru a-și schimba date, după care fiecare își continuă execuția în mod independent. Acest mod de sincronizare poartă numele de *randez-vous*. Se va prezenta în continuare cum este implementat acest concept în limbajul ADA.

Procesul apelant numește *procesul apelat* și punctul de intrare al acestuia. Procesul apelat, după ce a fost activat, constă din acțiuni secvențiale, ajungînd în anumite momente într-unul din punctele de intrare ce pot fi apelate din exterior. În momentul în care procesul apelant numește un punct de intrare în alt proces, pot exista 2 situații:

1) Procesul apelant nu a ajuns încă în punctul de intrare solicitat. În acest caz procesul apelant se suspendă și este pus în firul de așteptare asociat acelui punct de intrare. Cînd procesul apelant ajunge în acel punct de intrare, se va examina firul de așteptare și va fi servit procesul prioritar (primul sosit).

2) Procesul apelant a ajuns deja în punctul de intrare solicitat. Pînă la apariția unui apel extern, procesul apelant rămîne suspendat, avînd definit ca punct de reactivare punctul de intrare respectiv.

În ambele situații apelul constă din execuția acțiunilor asociate punctului de intrare în cauză, din cadrul procesului apelant, după care procesul apelant va fi reactivat și se va executa în paralel cu cel apelant.

În limbajul ADA fiecare proces este numit *task*. Definiția taskului constă din *specificația* taskului și din *corpul* acestuia. *Specificația unui task* precizează punctele de intrare

și eventualii parametrii asociați acestora. Într-un anume sens un punct de intrare poate fi înțeles ca o rutină. Spre exemplu, specificația:

```
task BUF is
   entry CITIRE (V:out ITEM);
   entry SCRIERE (E:in ITEM);
end;
```

este asociată unui task numit BUF, cu 2 puncte de intrare: CITIRE, cu parametrul de ieșire *V*, și SCRIERE, cu parametrul de intrare *E*.

Un alt task poate apela aceste puncte de intrare ca în exemplu următor:

BUF.SCRIERE(ELEMENT)

Corpul unui task conține marcat începutul acțiunilor asociate unui punct de intrare prin instrucțiunea *accept* urmată de numele punctului și parametrii săi. Acțiunile cuprinse în *randez-vous* sînt delimitate de *do* . . . *end*. Dacă la un moment dat în corpul unui task pot fi deschise mai multe puncte de intrare (care pot în acel moment să fie apelate) se poate folosi instrucțiunea *select*. Aceasta, împreună cu delimitatorul *end select*, cuprinde punctele de intrare care sînt simultan pregătite pentru apel. Este necesar însă să subliniem că în momentul în care unul din puncte este apelat, celelalte puncte simultan pregătite devin „pregătite" pînă la epuizarea acțiunilor legate de punctul considerat.

Iată spre exemplu corpul taskului BUF:

```
task body BUF is
   DIM: constant INTEGER: =20;
   BUFFER: array (1..DIM) of ITEM;
   IN, OUT: INTEGER range 1..DIM:=1;
   CONTOR: INTEGER range 0..DIM:=0;
begin
   loop
     select
       when CONTOR <DIM=>
         accept SCRIERE (E: in ITEM)
         do
             BUFFER (IN):=E
         end;
         IN:=IN mod DIM+1;
         CONTOR:=CONTOR+1;
     or
       when COMNTOR>0=>
       accept CITIRE (V: out ITEM)
         do
             V:=BUFFER(OUT);
         end;
           OUT:=OUT mod DIM+1
           CONTOR:=CONTOR-1
     end select;
   end loop;
end BUF;
```

Se observă că fiecare *accept* este precedat de o condiție, care nu este însă obligatorie; cînd se execută *select*, se evaluează condițiile tuturor punctelor de intrare și se consideră „deschise" toate punctele ale căror condiții sînt satisfăcute.

Excluziunea mutuală între taskuri este rezolvată prin 2 mecanisme. Primul este realizat de succesiunea în corpul taskului a instrucțiunilor *accept*, care se execută secvențial. Astfel, efectul unui semafor se poate exprima prin secvența:

```
accept P;
accept V;
```

Al doilea mecanism de excluziune mutuală este realizat prin *select*, deoarece am văzut că apelul unui punct deschis simultan cu altele va duce la închiderea celorlalte pe durata apelului.

Cooperarea are și ea două modalități de exprimare în ADA. Prima se bazează pe realizarea efectului de semafor, iar a doua este legată de punctele de intrare precedate de condiție într-un select.

S-a văzut că punctele de intrare la care condițiile nu sînt satisfăcute rămîn închise, procesele fiind puse în așteptare.

Însă apelul punctelor deschise conduce la modificarea variabilelor de care depind condițiile, astfel că la o nouă reevaluare unele din puncte se vor deschide. Procesele puse în așteptare la aceste puncte vor fi acum satisfăcute.

1.5.3. Planificarea proceselor

În existența sistemului de operare se ivesc multe situații cînd 2 sau mai multe procese sînt, din punct de vedere logic, executabile. Ca urmare, sistemul de operare trebuie să decidă pe care să-l activeze primul. Partea din sistemul de operare care are această sarcină se numește *planificator* iar algoritmul implementat în acest program se numește *algoritm de planificare*.

Exista cîteva criterii pe care trebuie să le satisfacă algoritmii de planificare:

— să fie sigur că fac să acceadă fiecare proces la unitatea centrală;

— să încarce unitatea centrală cît mai mult;

— să minimizeze timpul de răspuns pentru utilizatorii interactivi;

— să maximizeze numărul de programe prelucrate în unitatea de timp.

1.5.3.1 *Planificarea circulară*

Planificarea circulară („Round-Robin") reprezintă una din metodele de planificare cele mai răspîndite. Fiecărui proces îi este alocat pentru execuție un anumit interval de timp, numit *cuantă*. În momentul cînd un proces și-a epuizat cuanta, unitatea centrală este alocată altui proces. Dacă un proces se blochează (făcînd , de exemplu, o intrare/ieșire) înainte de epuizarea cuantei, sistemul de operare comută unitatea centrală în acest moment.

În vederea implementării acestei metode, planificatorul trebuie să rețină o listă cu procesele „gata". Cînd se epuizează o cuantă, procesul este pus la capătul listei, iar primul proces este activat (fig. 1.14).

Aspectul cel mai important îl reprezintă dimensiunea cuantei de timp. Comutarea de pe un proces pe altul cere timp și resurse — salvarea/reîncărcarea registrelor generale,

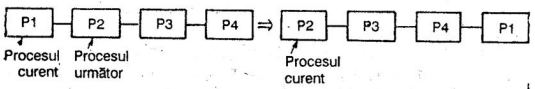


Fig. 1.14. Planificare circulară.

actualizarea unor structuri din baza de date a sistemului etc. Dacă presupunem, de exemplu, că procesul de comutare durează în medie 5 ms, o cuantă de 20 ms ar face ca 20% din timpul unității centrale să fie folosit pentru comutarea proceselor. Dacă cuanta crește la 500 ms, procentul se reduce la 1%.

1.5.3.2 *Planificator după priorități*

Planificarea circulară presupunea că toate procesele au aceeași prioritate. Frecvent însă utilizatorii au poziții diferite în ierarhie, fiind astfel nevoie de o planificare după priorități. Ca urmare , planificatorul va alege dintre toate procesele gata pe acelea care au prioritatea cea mai mare. Pentru a se evita ca procesele cu prioritate ridicată să se execute indefinit, planificatorul va descreste prioritatea procesului curent la fiecare cuantă de timp.

Prioritățile pot fi alocate static sau dinamic. De exemplu, într-o ierarhie rigidă, asemănătoare cu cea militară, prioritatea se alocă static, fix. În situația unor procese care fac operații de I/E lungi și așteaptă mult, acestea trebuie să aibă o prioritate ridicată în unitatea centrală. Ca urmare, se poate acorda dinamic o prioritate egală cu $1/n$, unde n este fracțiunea din ultima cuanta pe care procesul a folosit-o. Un proces care utilizează, de exemplu, doar 20 ms din cele 100 ms ale cuantei va avea prioritate 5, pe cînd unul care folosește 50 ms va avea prioritate 2. Se poate adopta și soluția grupării proceselor în clase; între clase se folosesc priorități, iar în interiorul claselor algoritmul de planificare este circular (fig. 1.15).

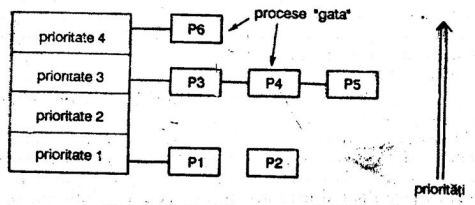


Fig. 1.15. Planificare cu priorități.

1.5.3.3. *Planificarea pe fire de așteptare multiple*

În cadrul acestei metode de planificare procesele sînt împărțite pe mai multe fire de așteptare. Procesele din clasa cea mai ridicată primesc cîte o cuantă de unitate centrală, cele din următoarea clasă, două cuante ș.a.m.d. Cînd un proces își consumă toate cuantele alocate va fi mutat în jos cu o clasă.

De exemplu, un proces, dacă are nevoie de 100 cuante pentru execuție, va primi pe rînd o cuantă, apoi 2, 4, 8, 16, 32, 64. Ca urmare, sînt necesare doar 7 întreruperi în loc de 100 în cazul algortimului de planificare circulară.

1.5.3.4 *Planificare uniformă*

Se poate pleca de la ideea că dacă există n utilizatori conectați, fiecare va primi aproximativ $1/n$ din timpul unității centrale. Sistemul trebuie să țină evidența timpului consumat de fiecare utilizator de la conectori. Apoi se calculează fracțiunea de timp, raportînd acest timp la timpul total la care avea dreptul (timpul total divizat cu n). Cunoscînd această valoare e ușor să se calculeze cît a folosit procesul unitatea centrală. Un raport de 1/2 arată că procesul a folosit doar jumătate din cît avea dreptul, iar un raport de 2 indică faptul că procesorul a utilizat de 2 ori mai mult decît avea dreptul. Algoritmul de planificare va face să se încarce procesele cu raportul cel mai mic.

1.5.3.5 *Planificarea pe 2 nivele*

Să presupunem că toate procesele „gata" sînt în memoria internă. Dacă însă spațiul de memorie devine insuficient, o serie din aceste procese sînt trecute pe disc. Această situație are implicație majoră pentru planificare, deoarece procesul de ecmutare crește foarte mult pentru procesele de pe disc.

O cale de contracarare a acestei dificultăți constă din a folosi o planificare pe 2 niveluri. Un subset din procesele „gata" este încărcat mai întîi în memoria internă. Planificatorul alege doar dintre aceste procese. Periodic un planificator de nivel înalt este apelat pentru a muta procese care

au stat mai mult timp în memorie și pentru a aduce altele de pe disc. Odată schimbarea făcută, planificatorul de nivel coborît alege din nou doar dintre procesele prezente în memoria internă!

1.6. GESTIUNEA INTRĂRILOR/IEȘIRILOR

1.6.1. Principii hardware de I/E

Una din cele mai importante funcții ale sistemului de operare o constituie controlul operațiunilor de intrare/ieșire cu perifericele din configurație. Acest lucru înseamnă comenzile perifericelor, tratarea întreruperilor și gestiunea erorilor, adică toate operațiile ce realizează interfața între periferice și restul sistemului de operare.

Perifericele de intrare/ieșire se împart în 2 categorii: bloc și caracter.

Perifericele bloc memorează informații în ansambluri de dimensiuni fixe, numite *blocuri*. Fiecare bloc are o adresă separată. Lungimea unui bloc este, în mod tipic, cuprinsă între 128 și 1 024 octeți. Una din proprietățile principale ale acestei categorii de periferice este că blocurile reprezintă unitate elementară de acces; ca urmare, scrierea sau citirea se face pe blocuri. În cazul driverelor, blocurile fizice sînt direct adresabile, existînd posibilitatea de a accesa orice bloc în orice moment; la benzi magnetice, deși informația este organizată tot pe blocuri, acestea nu sînt adresabile și accesul se face strict secvențial.

Periferice caracter, care acceptă sau furnizează șiruri de caractere, fără a vedea în ele structuri de blocuri. Din această categorie amintim terminalele, imprimantele, interfețele de rețea etc.

Sistemul de fișiere, componenta superioară a sistemului de operare, lucrează doar cu periferice bloc abstracte; în acest scop se folosește software dependent de periferic — așa numitele drivere.

Perifericele au în general 2 componente: o componentă mecanică — aparatul periferic propriu-zis și o componentă electronică — controlerul, de obicei inserat în calculator.

Controlerul (cuplorul) este cel cu care comunică sistemul de operare. De obicei mini şi microcalculatorul utilizează un model cu un singur BUS pentru comunicarea între unitatea centrală şi controler.

Sarcina controlerului este să convertească şirul de biţi în blocuri sau octeţi şi să facă eventualele corecţii ale erorilor. În general, controlerul înscrie la fiecare sector pe disc un preambul (la formatare, conţinînd numărul de cilindru, de sector şi dimensiunea sectorului) biţii propriu-zişi ai sectorului şi în final o sumă pentru controlul şi eventual corecţia erorilor (ECC). Blocurile sau octeţii sînt ansamblaţi/dezansamblaţi bit cu bit într-un buffer din controler. Apoi blocul este transferat în memoria internă sau cu perifericul.

Fiecare controler are un set de registre utilizate pentru comunicarea cu unitatea centrală. La o serie de calculatoare aceste registre fac parte din spaţiul memoriei interne. De exemplu, la minicalculatoarele compatibile PDP-11 există o pagină de intrare/ieşire între adresele 160 000-170 000.

Sistemul de operare face operaţiile de I/E, scriind comenzi în registrele controlerului. După ce comandă, parametrii fiind acceptaţi, unitatea centrală poate părăsi controlerul care lucrează independent. La sfîrşitul lucrului, controlerul generează o întrerupere pentru ca sistemul de operare să preia controlul şi să fie testat rezultatul operaţiei de I/E, prezent în registrele controlerului.

Perifericele-bloc, în mod special, folosesc — ca mod de acces — *accesul direct la memorie* (DMA — *D*irect *M*emory *A*cces). În cazul controlerelor fără DMA, după completarea bufferului cu informaţii de la periferic, se execută un ciclu prin care aceste informaţii se mută octet cu octet (sau cuvînt cu cuvînt) în memoria internă. Procesul prezentat este lent. În cazul controlerelor cu DMA, numai unitatea centrală are această sarcină; după ce controlerul a asamblat blocul în bufferul său, va muta primul cuvînt în memorie, la adresa specificată în DMA. Apoi va decrementa contorul, iar procesul *n* se repetă pînă ce contorul va ajunge la zero. Apoi controlerul va genera o întrerupere şi va anunţa sistemul de operare că transferul a fost terminat.

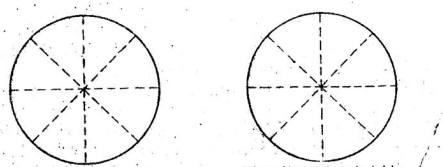


Fig. 1.16. Numerotarea blocurilor de disc.

Una din probleme se referă la motivul pentru care este nevoie de un buffer al controlerului şi nu se transferă direct informaţia în memorie. Dacă însă s-ar proceda aşa, transferul făcîndu-se prin BUS, poate apare situaţia în care BUS-ul să fie ocupat; în acest fel s-ar pierde informaţia de la periferic. Bufferul are rolul de a memora toată această informaţie, direct de la periferic.

În timpul în care datele sînt transferate din bufferul controlerului în memoria internă, sectorul următor se află deja sub capetele discului şi biţii săi sosesc la controler. Însă acesta nu poate face în acelaşi timp cele 2 operaţii şi sectorul este pierdut. Ca urmare, în funcţie de timpul de transfer necesar, se impune ca între citirea a 2 blocuri succesive să se sară unul sau mai multe sectoare fizice. Este deci raţional ca blocurile succesive să nu fie plasate în sectoare succesive, ci cu un *factor de salt*, numit *interleaving*. Cînd se formatează discul, blocurile sînt numerotate cu un astfel de factor de salt, ceea ce va permite sistemului de operare să poată citi blocuri succesive cu viteza maximă. În figura 1.16 se prezintă spre exemplificare modul cum sînt numerotate blocurile unei piste.

1.6.2 Principii software de I/E

Software-ul, de intrare/ieşire este organizat sub forma unor niveluri succesive, începînd cu cel mai de jos care ţine cont de particularităţile hardware-ului şi pînă la nivelul de sus care furnizează o interfaţă cît mai uniformă pentru utilizator.

1) Conceptul-cheie în proiectarea software-ului de I/E este *independența de periferic*. Trebuie, de exemplu, să fie posibil să se scrie un program care să lucreze, absolut identic, cu fișiere pe flopy-disc sau hard-disc.

2) O sarcină importantă a software-ului de intrare/ /ieșire o constituie *sincronizarea* la transferurile asincrone, gestionate prin întreruperi. Majoritatea perifericelor sînt asincrone; unitatea centrală startează transferul și apoi execută un alt program pînă la apariția întreruperii de sfîrșit de I/E. Programele utilizator sînt automat suspendate pînă ce datele sînt disponibile în buffer.

3) O altă sarcină o constituie *gestiunea perifericelor cu acces multiplu*, care pot fi folosite simultan de mai mulți utilizatori.

4) O sarcină importantă a software-ului de I/E o constituie tratarea și gestiunea erorilor care pot apare în lucrul cu perifericele.

Se pot distinge 4 niveluri în structura software-ului de I/E: gestiunea întreruperilor, driverele de periferice, soft-ul independent de periferic și soft-ul la nivel de utilizator.

Programele de I/E pot fi structurate pe 4 niveluri:

1) *Module de gestiune a I/E*, care au rolul să trateze întreruperile de I/E care sosesc de la periferice și să deblocheze driverele care sînt în așteptare;

2) *Driverele de periferice*, care înglobează codul dependent de periferic. Fiecare driver gestionează un anumit tip de periferic. Deoarece fiecare controler are un set de registre în care se dau comenzile, driverul va completa cu informații aceste registre. Driverul este modulul care are informații despre piste, cilindri, sectoare, factorul de salt; pe baza unor comenzi abstracte din partea nivelurilor superioare de I/E (independent de perifeoric), face să se declanșeze comenzi concrete de I/E.

După ce comenzile au fost completate în registre și lansate, fie că driverul așteaptă, blocîndu-se pînă sosește întrerupere de sfîrșit de I/E, fie că continuă execuția, cînd este vorba de anumite comenzi rapide. În ambele cazuri driverul va returna nivelurilor superioare informații de stare;

3) *Module software independente de periferic* ce au ca funcții tipice următoarele: interfața uniformă cu driverele,

gestiunea numelui de fișiere și de periferice, protecția accesului la periferice și fișiere, gestiunea bufferelor, alocarea spațiului disc și evidența alocării, raportarea erorilor. Aceste funcții sînt caracteristice sistemului de gestiune a fișierelor.

4) *Software-ul de I/E utilizator*, care se construiește pe baza unor primitive sisteme, link-editate împreună cu programele.

1.6.3 Blocarea proceselor la resurse

În cazul multiprogramării, o serie de procese care se execută în paralel utilizează un set finit de resurse. Dacă un proces cere o resursă, iar aceasta nu este disponibilă în acel moment, procesul intră într-o stare de așteptare. Dar s-ar putea ca procesul să nu mai iasă din această stare, deoarece resursa este deținută de un alt proces care se află în așteptare și el. De exemplu, să presupunem că în configurația unui calculator există doar două unități de bandă magnetică. Fie două procese dintre care fiecare deține la un moment dat cîte o unitate de bandă. Dacă fiecare proces are nevoie de 2 unități de bandă vor aștepta fiecare eliberarea perifericului de către celălalt proces. Aceasta este o *blocare*. Ori de cîte ori un proces obține accesul exclusiv la un periferic, fișier, articol sau resursă, există pericolul potențial de blocare.

Blocările apar atunci cînd procesele cer acces exclusiv la obiecte, numite *resurse*. Orice acces la o resursă presupune parcurgerea unei secvențe formate din 3 etape: cererea resursei, eliberarea și utilizarea ei, precum și eliberarea. Dacă resursa nu este disponibilă, procesul apelant se pune în așteptare; în multe sisteme de operare procesul se blochează automat cînd cererea eșuează și se deblochează cînd resursa devine disponibilă.

1.6.3.1 Modelarea blocărilor

Definiția formală a unei blocări poate fi următoarea: un set de procese se blochează dacă fiecare proces din mulțime așteaptă un eveniment pe care numai un alt proces

din mulțime îl poate realiza. Deoarece toate procesele sînt în așteptare, nici unul din ele nu realizează evenimentul; ca urmare, nici unul nu se va activa și procesele vor aștepta la infinit.

Coffman a stabilit cele 4 condiții necesare și suficiente să apară simultan pentru crearea unor blocări:

1) *Excluziune mutuală*: fiecare resursă este alocată la un singur proces sau este liberă. La un moment dat doar un singur proces poate utiliza resursa. Dacă un alt proces o cere, el va trebui pus în așteptare.

2) *Ocupare și așteptare*: procesele care ocupă resurse pot cere noi resurse. Dar aceste resurse pot fi deținute de alte procese.

3) *Eliberare*: resursele trebuie eliberate explicit de către procesele care le ocupă. Ele nu pot fi luate în mod forțat de la procese.

4) *Așteptare circulară*: poate exista un set de procese $\{p0, p1,..., pn\}$ în așteptare, astfel încît $p0$ așteaptă o resursă deținută de $p1$, $p1$ una deținută de $p2$,..., pn așteaptă o resursă ocupată de $p0$.

Blocările pot fi descrise mult mai precis prin utilizarea grafelor. Un graf constă dintr-o pereche $G=(V,E)$, unde V este o mulțime de noduri, iar E — o mulțime de arce. Mulțimea V se împarte în două clase: $P=\{p1,...pn\}$ — mulțimea proceselor și $R=\{r1,...rn\}$ — mulțimea resurselor. Fiecare element al lui E este o mulțime ordonată (p_i, r_i) sau (r_j, p_i), unde $p_i \in P$ și $r_j \in R$. Dacă $(p_i, r_j) \in E$ atunci este un arc direct de la procesul p_i la resursa r_j, implicînd că procesul p_i cere resursa r_j. Dacă $(r_j, p_i) \in E$, atunci există un arc de la resursa r_j la procesul p_i, implicînd că r_j a fost alocată procesului p_i. Perechea (p_i, r_j) este numită *arc-cerere*, iar perechea (r_j, p_i) — *arc-asignare*. Grafic, fiecare proces p_i se va reprezenta cu un cerc, iar fiecare resursă cu un dreptunghi. Dacă o resursă r este formată din mai multe exemplare, va conține în interiorul dreptunghiului atîtea puncte cîte unități de resursă de acel tip există. Să observăm că un arc-cerere va pointa către un dreptunghi, pe cînd un arc-asigurare va desemna un punct din interiorul pătratului.

Dacă un graf nu prezintă cicluri, nu poate produce blocări. În caz contrar, există situații cînd se pot produce blocări și situații cînd blocările nu sînt posibile.

Pentru evitarea blocărilor sistemul de operare poate adopta una din următoarele 4 strategii: ignorarea problemei, detecția și redresarea, prevenirea sau evitarea dinamică.

1.6.3.2. *Ignorarea blocărilor*

Această „tactică" reprezintă cea mai simplă abordare. Cum blocările apar rar, în timp ce căderile de sistem datorate unor pene hard sau anomalii soft sînt mult mai dese, se consideră că nu se justifică costurile în timp și soft pentru prevenirea acestora.

De exemplu, în sistemul de operare UNIX, unde nu se iau măsuri de detectare, blocările sînt posibile. De pildă, numărul total de procese în sistem este dat de numărul de intrări în tabela proceselor; cum aceasta este o resursă finită, la un moment dat se poate umple, iar programul care execută *fork* va aștepta. Dacă toate procesele în execuție execută *fork* dorind să creeze subprocese, iar tabela de procese este plină, sistemul intră în blocare. Posibilitatea, deși minimă, există. Blocări similare la UNIX pot apărea și în ceea ce privește spațiul de evacuare sau numărul de fișiere. Cum însă ele apar foarte rar, se preferă să nu se încarce sistemul de operare cu mecanisme complicate de detecție a blocărilor.

1.6.3.3. *Detecția blocărilor*

În aceste cazuri sistemul de operare monitorizează cererile și eliberările de resurse. În toate aceste momente se actualizează graful alocării resurselor și se verifică dacă există cicluri. Dacă da, unul din procesele din ciclu este distrus și astfel ciclul este spart.

1.6.3.4. *Prevenirea blocărilor*

Această metodă presupune impunerea unor restricții proceselor, astfel încît blocările să fie imposibile. Cele 4 condiții relevate de Coffman reprezintă punctul de plecare:

dacă ne putem asigura că cel puțin una dintre ele nu este niciodată satisfăcută, blocările sînt imposibile.

**Excluziunea mutuală.* Dacă nici o resursă nu este alocată exclusiv unui proces, nu pot apare blocări. Dar este evident că anumite resurse (de exemplu, imprimanta) nu pot fi folosite simultan de mai multe procese. O soluție este de a folosi imprimanta în „spooling", cînd un singur proces care are acces fizic la ea („daeman"). Din nefericire nu orice periferic sau resursă poate fi utilizată în spool. În plus, competiția pentru spațiul de pe disc al fișierelor spool poate ea însăși conduce la blocări.

**Ocupare și așteptare.* Este necesar să se prevină faptul că procesele care ocupă resursele să aștepte pentru alte resurse. Un mod de a rezolva această cerință este de a pretinde tuturor proceselor să ceară toate resursele de care au nevoie la început, înainte de lansarea în execuție. Dacă ele sînt libere, procesul este lansat și se execută fără probleme pînă la sfîrșit.

O altă soluție este de a cere proceselor care au nevoie de o resursă să elibereze mai întîi în mod temporar toate resursele pe care le deține și apoi să le ceară din nou, împreună cu resursa nouă.

**Eliberare.* Dacă un proces care deține resurse cere alte resurse ce nu pot fi alocate imediat, atunci el va elibera toate resursele deținute. Acestea, împreună cu resursele noi cerute, vor figura în lista de așteptare a procesului. Ideea, greu de implementat în practică, este ca procesele să nu se blocheze deținînd resurse pe care le cer alte procese ce se află în așteptare.

**Așteptare circulară.* Această stare se poate elimina prin mai multe metode:

a) Un proces va lucra la un moment dat cu o singură resursă. Dacă are nevoie de a doua trebuie să o elibereze pe prima. Însă astfel de condiții sînt pentru unele resurse inacceptabile;

b) Se poate face o numerotare globală a tuturor resurselor printr-o funcție $F: R \rightarrow N$, de exemplu:

F (cititor cartele)=1
F(disc)=5

F(banda)=7
F(imprimanta)=12

Regula este următoarea: procesele pot cere resurse cînd doresc, dar toate cererile trebuie făcute în ordine numerică. De exemplu, dacă un proces cere r și r în această ordine, va trebui ca $F(r) > F(r)$. Cu această regulă, graful alocării resurselor nu va avea niciodată cicluri. Vom demonstra prin reducere la absurd. Fie procesele $\{p0, p1, ..., pn\}$ într-un ciclu, unde p așteaptă resursa r, deținută de p. Dar acest lucru implică secvența următoare de inegalități:

$$F(r0) < F(e1) < .... < F(rn) < F(r0), \quad \text{adică} \quad F(r0) < F(r0),$$

ceea ce este imposibil. Ca urmare, nu poate să apară așteptare circulară.

1.6.3.5. *Evitarea blocărilor*

Evitarea blocărilor se poate face doar dacă se dispune de unele informații în avans. Vom prezenta cîțiva din algoritmii cei mai cunoscuți:

a) *Algoritmul bancherului pentru o singură resursă*

Se încearcă modelarea modului în care lucrează un bancher cu un set de creditori, cărora le-a împrumutat sume de bani. Prin extensie fie exemplul alocării unei cantități de t unități de bandă magnetică între n utilizatori. Fiecare utilizator specifică în avans numărul maxim de unități de bandă de care are nevoie în timpul execuției. Sistemul de operare va accepta cererile utilizatorilor atîta timp cît ele nu vor depăși t. Fiecare utilizator poate obține sau elibera pe rînd o bandă. În cazul cererii, utilizatorul poate aștepta dacă nu există unități libere, dar sistemul de operare garantează că timpul de așteptare este finit.

O stare curentă este *sigură* dacă pentru sistemul de operare este posibil să permită că fiecare utilizator să termine într-un timp finit. În caz contrar, starea este *nesigură*.

Fie n utilizatori și $l(i)$ numărul curent de benzi alocate utilizatorului i. De exemplu, dacă $l(5)=4$, au fost alocate pînă în acest moment 4 benzi utilizatorului 5. Fie $m(i)$ cererea maximă de benzi pentru utilizatorul i. Fie $c(i)$

cererea maximă de benzi pentru utilizatorul i. Fie $c(i)$ cererea disponibilă a utilizatorului i, $\mathbf{c(i)=m(i)-l(i)}$.

Sistemul de operare controlează alocarea celor t benzi. Fie a numărul de unități disponibile la un moment dat pentru alocare.
Se observă că:

$$a = t - \sum_{i=1}^{n} l(i).$$

Algoritmul bancherului propus de Dijkstra pleacă de la premisa că se va face alocare doar atunci cînd se va intra într-o stare sigură, adică într-una în care situația resurselor permite tuturor utilizatorilor să termine programele.

De exemplu, în figura 1.17*a* se prezintă un exemplu de *stare sigură*. Să presupunem că există 12 unități de bandă și 3 utilizatori care le folosesc, aceștia fiind u_1, u_2, u_3.

Starea (*a*) este sigură doarece este posibil pentru toți cei 3 utilizatori să termine. Utilizatorul u_2 are alocate 4 benzi și cere maxim 6. Deci cele 2 pe care le poate eventual cere pot fi satisfăcute din cele 2 disponibile în sistem. Dacă benzile sînt alocate lui u_2, acesta va trebui să se execute pînă se termină, moment în care va elibera toate cele 6 benzi alocate. O parte din ele pot fi alocate lui u_1 (maxim 3), altele lui u_3 (maxim 3), care acum pot să-și continue lucrul. Ca urmare, există o cale de a termina pentru toți utilizatorii.

În figura 1.17 b se prezintă o *stare nesigură*: 11 din cele 12 benzi sînt alocate și numai una este disponibilă. Continuarea ulterioară a programelor u_1, u_2, u_3 nu garantează

	c(i)	m(i)
u_1	1	4
u_2	4	6
u_3	5	8
disponibil	2	

(a)

	c(i)	m(i)
u_1	8	10
u_2	2	5
u_3	1	3
disponibil		

(b)

	c(i)	m(i)
u_1	1	4
u_2	4	6
u_3	5	8
disponibil	2	

(c)

	c(i)	m(i)
u_1	1	4
u_2	4	6
u_3	6	8
disponibil	1	

(d)

Fig. 1.17. Algoritmul bancherului cu o resursă.

că toate 3 vor ajunge să termine. Dacă, de exemplu, u_1, u_2 și u_3 vor cere cîte 2 benzi, fără a elibera nici una, apare o blocare.

În figura 1.17*c* se prezintă o stare sigură care însă, printr-o tranziție, va conduce la o stare nesigură (fig. 1.17*d*). De exemplu, în (*c*) u_3 va cere încă o unitate de resursă, intrîndu-se în starea (*d*) care poate duce la blocare.

În general, algoritmul bancherului trebuie să analizeze fiecare stare apărută în urma unei cereri dacă este sigură sau nu. Dacă este sigură, cererea va fi satisfăcută; în caz contrar, va fi amînată. Pentru a vedea dacă o stare este sigură se va încerca să se determine dacă există resurse pentru a satisface programul apelant cu cererea sa maximă care poate să apară. În caz afirmativ, se consideră alocarea făcută, și cu noua valoare disponibilă se face verificare la noul program ș.a.m.d. Dacă toate programele pot fi satisfăcute, starea este sigură.

b) *Algoritmul bancherului pentru resurse multiple*

Se poate încerca generalizarea algoritmului lui Dijkstra pentru o resursă în cazul existenței mai multor tipuri de resurse: de exemplu, 3 benzi, 2 plottere etc.

Așa cum se vede în figura 1.18 se folosesc 2 matrice: cea din stînga (*a*) indică resursele asigurate, $l(i)$, pentru fiecare program utilizator. Cea din dreapta (*b*) indică cîte

Procese	tp	tpr	pl
u_1	3	0	1
u_2	0	1	0
u_3	1	1	1
u_4	0	0	0
u_5	0	0	0

(a) Resurse alocate

Procese	tp	tpr	pl
u_1	1	1	0
u_2	0	1	1
u_3	3	1	0
u_4	0	0	1
u_5	2	1	1

(b) Resurse necesare

E = (6, 3, 4)
P = (5, 3, 2)
A = (1, 0, 2)

Fig. 1.18. Algoritmul bancherului cu resurse multiple.

resurse sînt necesare fiecărui proces pentru ca acesta să se termine, $c(i)$. Toate procesele vor trebui să indice la început valorile $m(i)$ pentru cererile din fiecare tip de resursă. Se folosesc și 3 vectori: E — cu resursele existente la fiecare tip, P — cu resursele posedate (alocate) din fiecare tip și A — cu resursele disponibile. De exemplu, $E=(6,3,4)$ indică faptul că există 6 benzi, 3 imprimante și 4 plottere.

Algoritmul pentru verificarea proprietății de „sigur" a unei stări constă din următorii pași:

(1) Se analizează un rînd R la care necesitățile de resurse sînt mai mici decît vectorul A. Dacă nu există un astfel de rînd se intră în blocare;

(2) Să presupunem că procesul de pe rîndul găsit cere toate resursele de care are nevoie și termină. Se marchează această stare, resursele devenite disponibile adunîndu-le la vectorul A;

(3) Se repetă pașii (1) și (2) pînă cînd toate procesele sînt marcate ca terminate. În acest caz starea este sigură.

Algoritmul este foarte interesant, însă greu de aplicat deoarece nu toate procesele pot cunoaște în mod apriori necesarul de resurse.

1.7. SISTEMUL DE FIȘIERE

Pentru majoritatea utilizatorilor, sistemul de fișiere reprezintă partea cea mai vizibilă a sistemului de operare. Fișierele memorează în general date și programe. Sistemul de operare implementează conceptul de fișier, gestionînd în mod corespunzător perifericele de tip disc sau bandă.

1.7.1 Conceptul de fișier

Calculatoarele pot memora informația în diferite forme fizice: pe bandă, pe disc etc. Fiecare din aceste periferice are propriile lui caracteristici și organizări fizice. Pentru a facilita utilizarea sistemelor de calcul, sistemele de operare furnizează o modalitate uniformă de a vedea din punct de vedere logic informația memorată. Se definește astfel o unitate logică de memorare, numită *fișier*. Sistemul de operare este cel care asigură corespondență fiecărui fișier cu un periferic fizic.

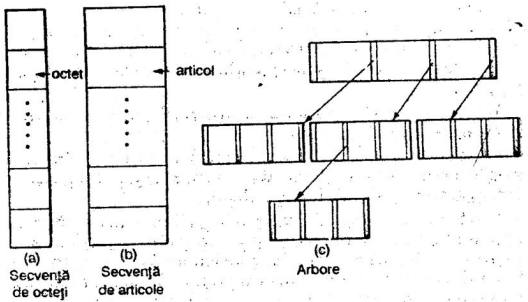


Fig. 1.19. Moduri de organizare a fișierelor.

Un fișier este o colecție de informații — programe (în format sursă sau obiect) sau date. Sistemul de operare pune la dispoziția utilizatorilor operații speciale, numite apeluri sistem, în vederea creării sau distrugerii fișierelor, pentru citirea sau scrierea de informații din/în ele precum și pentru alte nevoi de gestiune.

După cum rezultă din figura 1.19 există trei *moduri de organizare a fișierelor*:

(*a*) Ca o simplă secvență de octeți (ca în UNIX);

(*b*) Ca o secvență de articole cu lungime fixă; orice articol poate fi citit sau scris, dar nu pot fi inserate sau șterse articole în mijlocul fișierului (ca în CP/M);

(*c*) Ca un arbore de blocuri disc, fiecare bloc conținînd un număr de articole cu chei. Articolele pot fi localizate prin cheie și se pot insera articole oriunde în arbore. Dacă se adaugă un nou articol la un bloc care este plin, acesta este spart în 2 blocuri, fiecare dintre ele fiind introdus în arbore. O astfel de metodă se numește *secvențial — indexată*.

Operațiile permise asupra fișierelor diferă de la un sistem de operare la altul. Citirea/scrierea de octeți (articole) în modul *secvențial* este totdeauna permisă. Unele sisteme de operare permit însă și accesul *aleatoriu* (direct), cînd se specifică cheia sau numărul articolului. În alte sisteme de

operare, cum ar fi UNIX, se poate seta oriunde poziția curentă în fișier, astfel încît noile operațiuni de citire-/scriere vor apărea în noua poziție. Sistemele de fișiere orientate pe articole mai permit operații de *inserare* sau *ștergere*.

Orice sistem de operare furnizează cel puțin următoarele 5 *operații asupra fișierelor*:

1) *Creare fișiere*. În vederea creării unui fișier sînt necesare 2 etape: găsirea de spațiu pentru fișier (alocarea) și înregistrarea fișierului în director;

2) *Scrierea în fișier*. Pentru a scrie într-un fișier, trebuie specificate numele fișierului și informația care a fost scrisă pe disc. Sistemul de operare, plecînd de la nume, va căuta fișierul în director, de unde va extrage localizarea acestuia (adresă de început). Pe baza acestei informații, se calculează adresa blocului de la care se va face scrierea. Prin incrementarea acestei adrese, se pot apoi scrie, succesiv, secvențe de blocuri în fișier.

3) *Citire în fișier*. Pentru a citi într-un fișier, trebuie specificate numele fișierului și adresa de memorie unde se va pune informația citită. Din nou se caută numele fișierului în director, se extrage localizarea acestuia și se calculează adresa bloc la care se face citirea.

În general la un moment dat, un fișier poate fi scris sau citit fără ca operațiile de scriere/citire să se întrepătrundă;

4) *Rebobinare fișiere* implică poziționarea fișierului la începutul său;

5) *Ștergere fișier*. Pentru a șterge un fișier, se caută în director, se eliberează zona alocată fișierului și apoi se invalidează intrarea corespunzătoare în director.

1.7.2. Fișiere bandă și disc

Fișierele se pot memora pe bandă magnetică sau disc. *Fișierul bandă* are avantajul simplității și al ușurinței păstrării suporților. Însă, la fișiere mici, rămîne o bună parte din bandă nefolosită. De asemenea, banda nefiind un suport direct adresabil, orice acces la un articol presupune parcurgerea secvențială a tuturor articolelor precedente. Acest lucru face ca banda să fie un suport lent, preferat pentru

arhivarea unor date sau programe. Pentru fișierele foarte mari se pot crea stocări *multivolum*.

Pentru a rezolva problema utilizării eficiente a suportului bandă, se pot memora mai multe fișiere mici pe același suport, organizarea fiind în acest caz *multifișier — monovolum*.

O altă dificultate constă în modul în care se actualizează unele articole din fișierele bandă. Deoarece nu se poate scrie exact peste vechiul articol, trebuie recopiate toate articolele pe o altă bandă, unde se creează astfel fișierul modificat. Uneori se adaugă la începutul benzii un *director*, care să reținâ numele fișierelor memorate pe suport.

O altă categorie, mult mai eficient de utilizat, este aceea a fișierelor disc. Din punct de vedere structural, un disc se divide în cilindri, piste și sectoare. Unitatea de bază care poate fi citită sau scrisă pe disc este *sectorul*, a cărui dimensiune poate varia (tipic 512 sau 1 024 octeți). Pe fiecare față a discului sînt mai multe piste, fiecare din ele divizată în sectoare. Capetele de citire/scriere, sînt deplasate pe pista adresată, apoi prin rotire se ajunge la sectorul căutat. Unele discuri au mai multe suprafețe; sectoarele tuturor suprafețelor care au aceeași poziție formează un cilindru.

Operațiile de I/E între memorie și disc sînt efectuate în unități de unul sau mai multe sectoare. Adresarea unui anumit sector implică precizarea cilindrului (sau pistei), feței și sectorului. Ca urmare, un disc poate fi tratat de către sistemul de operare ca un tablou unidimensional de *blocuri disc*. Fiecare bloc este un sector. Dacă notăm cu s numărul de sectoare de pe o pistă și cu t numărul de piste pe cilindru, atunci se poate converti o adresă disc de forma (i,j, k), unde i este adresa cilindrului, j — adresa suprafeței și k — adresa sectorului, într-un număr de bloc, $n = k+s\ (j + i \times t)$.

Din punct de vedere tehnologic și al capacității de stocare există mai multe tipuri de discuri: floppy, cu dimensiuni 5" și 1/4 sau 3" și 1/2, capacitatea lor fiind de 512 Kocteți 1 Moctet și 1,2 Mocteți; discuri Winchester, foarte compacte și cu capacități mari (de la 10 Mocteți pînă la sute sau mii de Mocteți); discuri amovabile, specifice minicalculatoarelor, cu capacități de zeci sau sute de Mocteți.

1.7.3. Directoare

Directoarele au rolul de a ține evidență fișierelor și a informațiilor referitoare la acestea: nume, dimensiune, proprietar, tip etc. În unele sisteme de operare, directoarele sînt și ele memorate sub forma unor fișiere. Fiecare intrare într-un director este asociată unui fișier și conține toate datele necesare prelucrării acestuia prin intermediul sistemului de fișiere.

Putem spune că directorul reprezintă o tabelă care translatează numele de fișiere în numere corespunzătoare intrărilor în director. Ca urmare, organizarea directoarelor se poate face în mai multe moduri. Cea mai simplă cale pentru sistemul de operare este să rețină *un singur director* care să conțină toate fișierele. O atare soluție este însă greoaie, lentă și generatoare de confuzii, deoarece mai mulți utilizatori pot da același nume la fișiere diferite (soluția se regăsește, de exemplu, în sistemul de operare CP/M).

O altă soluție este să existe cîte un director afectat fiecărui utilizator — *2 niveluri de directoare*, În acest caz arborele sistemului de fișiere are un nivel în plus și se asigură o mai bună comportare între utilizatori (soluție adoptată, de exemplu, la RSX-11M).

În fine, o altă soluție constă în crearea unei *ierarhii arborescente de directoare* într-un arbore cu un număr nelimitat de niveluri și care conține drept noduri fișiere sau directoare (soluție implementată în MS-DOS și în UNIX). Localizarea unui fișier într-un astfel de arbore se face fie printr-o *cale absolută*, care pleacă din directorul rădăcină, fie printr-o *cale relativă*, care începe cu subdirectorul curent. Fiecare utilizator are afectat un director care este *director curent*.

Fiecare director (subdirector) conține o mulțime de fișiere și/sau alte subdirectoare. Cîte un bit în fiecare intrare a unui director diferențiază fișierele de subdirectoare.

O altă soluție de organizare a sistemului de fișiere constă într-un *graf aciclic de directoare*. Este cazul în care anumite subdirectoare sau fișiere sînt folosite în comun. Acestea există în arborele sistemului de fișiere în mai multe locuri deși fizic, pe disc, se află memorate o singură dată.

Un astfel de graf aciclic (fără cicluri) reprezintă o generalizare naturală a schemei de tip arbore și este utilă atunci cînd mai mulți utilizatori lucrează în echipe. Implementarea unei astfel de structuri se poate face în mod diferit. De exemplu, se pot crea în directoare *intrări de tip legătură*, care furnizează un pointer către altă intrare de director ce conține informații despre fișierul respectiv.

O altă posibilitate de implementare constă în duplicarea tuturor informațiilor despre fișiere în ambele directoare. O problemă mai dificilă se întîlnește la ștergerea fișierelor și directoarelor partajate, deoarece ștergerea unei legături nu trebuie să implice eliberarea zonei de disc afectată fișierului, întrucît acesta mai poate fi folosit de alți utilizatori prin intermediul altor legături.

În fine, o ultimă soluție de organizare a sistemului de fișiere o reprezintă *graful general de directoare*. Dacă în organizarea arborescentă a sistemului de fișiere se vor introduce legături fără restricția de a avea cicluri, se obține o structură de graf. În astfel de cazuri, însă, algoritmii de căutare în structură, de localizare a unui fișier, precum și cei de ștergere devin mai complicați, ei trebuind să evite cicluri infinite.

1.7.4. Alocarea spațiului disc

Din punctul de vedere al utilizatorului, fișierele sînt un tip de date abstracte. Ele pot fi create, deschise, scrise, citite, închise, șterse etc., fără nici o referire reală la implementarea acestora pe suportul magnetic. Implementarea fișierelor reprezintă o problemă a sistemului de operare.

Fișierele sînt cel mai frecvent, memorate pe disc. În gestiunea discului sînt posibile 2 strategii: fie să se aloce toți cei n octeți a unui fișier în mod consecutiv, fie să se „spargă" fișierul într-un număr de blocuri care nu trebuie neapărat să fie memorate consecutiv pe disc.

1.7.4.1. *Dimensiunea blocurilor*

Una dintre problemele fundamentale în proiectarea sistemului de fișiere este cît de mare trebuie să fie blocul. Desigur, acesta poate să fie un multiplu întreg de sectoare, piste sau cilindrii.

Dacă unitatea de alocare este mare (de exemplu, un cilindru) orice fișier, chiar dacă ar avea un singur octet, va ocupa cel puțin un cilindru. Studiile întreprinse la UNIX au arătat, de exemplu, că dimensiunea medie a unui fișier este de 1 Koctet. Dacă unitatea de alocare este cilindru (32 Kocteți), se pierd aproximativ 97% din spațiul disc. În situația că unitatea de alocare este mică fiecare fișier va avea un număr mare de blocuri și citirea lui va fi mai lentă; pentru fiecare bloc se face poziționarea cu așteptare, în medie, a unei rotații.

În figura 1.20 curba (*1*) indică timpul de acces la date (viteza de obținere a acestora), iar curba (*2*) reprezintă procentul de utilizare a discului în funcție de dimensiunea blocului. Din studiul acestor grafice rezulta o *valoare de compromis* pentru dimensiunea blocului, de 512 octeți, 1 Koctet, 2 Kocteți, care asigură atît o bună utilizare a discului, cît și o viteză acceptabilă de acces la date. Dacă se alege dimensiunea blocului de 1 Koctet, sistemul de fișiere va citi/scrie cîte două sectoare la fiecare acces.

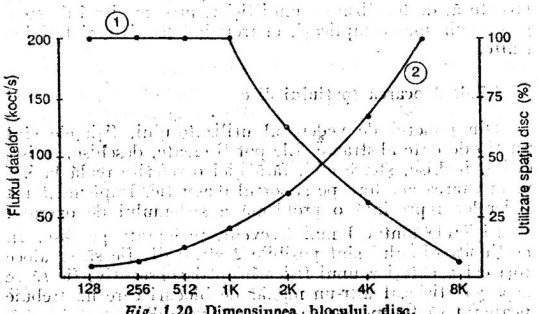


Fig. 1.20. Dimensiunea blocului disc.

1.7.4.2. *Evidența spațiului liber*

Fișierele sînt create și șterse cu o mare frecvență în cadrul unui sistem de operare. Ca urmare, trebuie să existe o evidență permanentă a spațiului disc disponibil, care să memoreze toate *blocurile libere* nealocate nici unui fișier. La crearea unui fișier se caută un număr de blocuri libere, iar la ștergerea unui fișier tot spațiul alocat acestuia este returnat zonelor libere.

Evidența acestui spațiu liber se poate face în principal prin 2 metode:

(*1*) Utilizarea unei *liste înlănțuite* a blocurilor disc libere, fiecare bloc reținînd mai multe numere de blocuri libere. Dacă, de exemplu, blocul are 1 Koctet, iar numărul de bloc are 16 biți, rezultă că fiecare bloc din listă poate reține numerele a 512 alte blocuri libere. Un disc de 20 Mocteți are nevoie pentru lista blocurilor libere de 40 de blocuri pentru toate cele 20 000 de unități de alocare;

(2) Se poate utiliza o *hartă de alocare*, preferabil parțial în memoria internă, în cadrul căreia se afectează fiecare bloc cîte un bit. Un disc de n blocuri are nevoie de o tabelă de n biți, în care cele libere sînt marcate cu 0, iar cele ocupate cu 1. Ca urmare, un disc de 20 Mocteți, cu 20 000 de blocuri are nevoie doar de 3 blocuri de hartă de alocare.

1.7.4.3. *Evidența spațiului alocat*

Un fișier constă dintr-o succesiune de blocuri. Sistemul de fișiere trebuie să țină evidența blocurilor care aparțin fiecărui fișier. Din punctul de vedere al alocării zonelor disc pentru fișiere deosebim 3 moduri de abordare: alocare continuă, alocare înlănțuită și alocare indexată.

1.7.4.3.1. Alocarea contiguă

Metoda de alocare contiguă cere ca fiecare fișier să ocupe un set contiguu de adrese pe disc. Adresele disc definesc o secvență liniară.

Alocarea contiguă a unui fișier este definită de adresa primului bloc și de lungime. Dacă un fișier are n blocuri și începe cu adresa b, el va ocupa blocurile b, $b + 1$, $b + 2$, ...

..., $b + n - 1$. Intrarea în director corespunzătoare fiecărui fișier indică adresa de început a fișierului și lungimea.

Accesarea unui bloc în acest caz este foarte simplă și în acces secvențial, precum și în acces direct. Dificultatea în cazul acestui mod de alocare constă în găsirea spațiului pentru un nou fișier. De exemplu, într-o hartă cu biți care să reprezinte cu 1 blocurile alocate și cu 0 pe cele libere, va trebui să găsim n de 0 succesiv.

Problema poate fi tratată ca un caz particular de *alocare dinamică*; spațiul disc se poate asimila cu o matrice de blocuri În orice moment unele blocuri pot fi alocate, altele libere. Spațiul disc poate fi văzut ca o mulțime de segmente; fiecare segment reprezintă o mulțime de blocuri. Unele segmente sînt alocate, altele sînt libere, acestea numindu-se *goluri*.

Alocarea dinamică constă în a satisface o cerere de n blocuri dintr-o listă de goluri. Există mai multe posibilități:

a) „Prima potrivire" — cînd se alocă din primul gol întîlnit în lista care are dimensiune mai mare sau egală cu n;

b) „Cea mai bună potrivire" — cînd se alocă din golul cel mai mare al listei care are dimensiunea mai mare sau egală cu n;

c) „Cea mai proastă potrivire" — cînd se alocă din golul cel mai mare al listei, care are dimensiunea mai mare sau egală cu n.

Desigur că în toate aceste metode de alocare apare o *fragmentare* a spațiului disc în goluri din ce în ce mai mici. De aceea, în anumite momente, va fi necesară o concatare a golurilor — *compactare* — care se realizează prin mutarea fișierelor unul după altul.

1.7.4.3.2. Alocarea înlănțuită

Prin memorarea blocurilor alocate fișierelor ca o listă înlănțuită, dispare necesitatea găsirii unui spațiu contiguu pentru fișiere. În acest caz, directorul conține un pointer către primul (sau ultimul) bloc al fișierului. Fiecare bloc la rîndul său va conține cîte un pointer către următorul bloc.

Acești pointeri sînt invizibili pentru utilizatori. În aceste cazuri crearea unui fișier implică doar crearea unei intrări în directorul al cărui pointer către primul bloc este inițializat cu *nil* (fișier gol). La prima scriere, în fișier se va căuta un bloc din lista spațiului liber care va fi pointat de către informația din director.

Avantajul acestei metode de alocare constă în aceea că nu produce o fragmentare a spațiului disc. Orice bloc liber oriunde ar fi plasat, poate fi alocat unui fișier. Dezavantajul principal al metodei este acela că o astfel de alocare permite doar acces secvențial, deoarece trebuie parcursă lista de pointeri. Însă accesul la un pointer implică citirea unui bloc, ceea ce mărește considerabil timpul de acces la un anumit bloc. Un alt dezavantaj constă în spațiul ocupat de pointeri, care reduce cu cîteva procente spațiul efectiv folosit pentru informații.

1.7.4.3.3. Alocare indexată

Pentru a se rezolva problema accesului direct, toți pointerii pot fi aduși la un loc, într-un *bloc indexat*. Fiecare fișier are propriul său bloc index, care este o matrice de adrese de blocuri disc. Intrarea i în blocul index conține pointerul către cel de-al i-lea bloc al fișierului. Pentru a citi direct blocul i, se va utiliza intrarea i din blocul index și apoi se va citi propriu-zis blocul dorit. Cînd se creează un fișier toți pointerii din blocul index sînt inițializați cu *nil*.

Și în cazul alocării indexate, fragmentarea nu mai reprezintă o problemă, deoarece blocurile pot fi alocate oriunde pe disc. Alocarea indexată are dezavantajul consumării unui spațiu destul de important cu blocurile index. Aceasta cu atît mai mult cu cît majoritatea fișierelor sînt mici și, ca urmare, blocurile index sînt neeficient folosite. De aici derivă problema dimensiunii optime a blocurilor index, care trebuie analizate de la implementare la implementare. După cum se va vedea în cazul sistemului de operare UNIX, se pot adopta soluții mai complexe, cu mai multe niveluri de blocuri index, dintre care doar unele să fie prezente în memoria internă după deschiderea fișierului.

1.8. GESTIUNEA MEMORIEI

Gestiunea și organizarea memoriei interne reprezintă unul din cei mai importanți factori care influențează proiectarea și performanțele sistemelor de operare. Acestea trebuie să țină evidența spațiului de memorie, zonelor libere sau ocupate, să aloce memorie la procese sau să elibereze zone deja alocate, să realizeze operațiunile de evacuare/încărcare a proceselor din/în memoria internă.

1.8.1. Organizarea memoriei

Memoria internă a fost considerată mult timp ca o resursă foarte scumpă. De aceea, o bună parte din efortul de proiectare a sistemului de operare a vizat utilizarea eficientă a acestei resurse. În general, propunerile și datele necesare execuției sînt ținute în *memoria internă*. Cele care nu sînt necesare imediat pot fi memorate pe o *memorie secundară*, de obicei disc. Se poate vorbi deci de o *ierarhie de memorii*. (figura 1.21), în care ulterior s-a mai introdus un nivel — *memorie cache* — cu influențe favorabile asupra performanțelor de timp ale sistemelor. Această memorie este mai rapidă decît memoria internă, mai scumpă, dar și mai mică ca dimensiuni. Ea va reține datele și zonele de program cel mai recent folosite de unitatea centrală, care de obicei au șanse mari de a fi referite în continuare din nou. La următoarea lor utilizare, accesul se va face mult mai rapid, el găsindu-se în memoria cache.

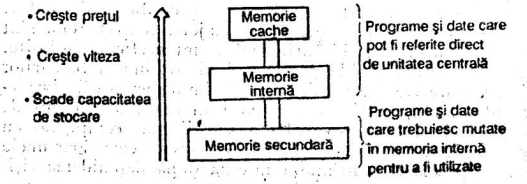


Fig. 1.21. Ierarhia organizării memoriei.

Fig. 1.22. Monoprogramare.

De obicei, memoria internă este împărțită în mai multe programe în zone numite *partiții*. În strategia de proiectare a gestiunii memoriei, acele partiții pot fi făcute fixe sau variabile, cu o dimensiune egală cu a proceselor în execuție.

Există sisteme de operare care, din dorința de a servi simultan mai mulți utilizatori, evacuează temporar, pe rînd, anumite programe de disc, pentru a face loc altora în memoria internă. Aceste sisteme folosesc tehnica de *evacuare/încărcare*, numită și *swapping*. Alte sisteme de operare mai sofisticate folosesc conceptul de *memorie virtuală*, care implică prezența programelor în memoria internă, parțial în memoria externă. Toate aceste concepte vor face obiectul unei prezentări în continuare.

1.8.2. Monoprogramarea

Cea mai simplă schemă de gestiune a memoriei interne este aceea de a avea la un moment dat *un singur proces în memoria internă*, căruia i se alocă toată memoria internă disponibilă. În figura 1.22 sînt prezentate 3 variante utilizate încă pe unele microcalculatoare sau calculatoare personale. Memoria internă este împărțită între sistemul de operare și utilizator. Sistemul de operare poate fi la începutul memoriei, în RAM (*a*), la sfîrșitul memoriei, în ROM (*b*), sau se poate face o repartiție a driverelor ca în (*c*).

De exemplu, la IBM-PC sistemul de operare MS-DOS folosește schema (*c*) cu driverele în ROM: ultimii 8 Kocteți din 1 Moctet spațiu de adrese. Programul din ROM se numește BIOS.

În cazul acestor sisteme de operare se execută la un moment dat un singur proces cerut de utilizator la tastatură. Cînd procesul se termină, sistemul de operare afişează un prompter pe ecran şi aşteaptă o nouă comandă care implică încărcarea altui proces.

1.8.3. Multiprogramarea cu partiţii fixe

Execuţia doar a unui singur program în memoria internă duce la o utilizare necorespunzătoare a unităţii centrale care aşteaptă perioade foarte lungi cît timp programul execută operaţii de intrare/ieşire, mult mai lente. De aici ideea de a folosi mai intensiv unitatea centrală prin execuţia altui program în timp ce primul aşteaptă sfîrşitul unor operaţii de intrare/ieşire (*multiprogramare*).

Cea mai simplă cale de asigurare a multiprogramării este de a împărţi memoria internă în *n* partiţii, posibil inegale. Cînd soseşte un program, el este pus în firul de aşteptare al celei mai mici partiţii în care el încape (vezi figura 1.23).

Întrucît însă partiţiile sînt fixe, se pierde spaţiu în fiecare partiţie, programele fiind mai mici ca dimensiune. Odată cu introducerea conceptului de multiprogramare trebuie rezolvate 2 probleme: relocarea şi protecţia.

Relocarea presupune ca programul să fie astfel compilat şi link-editat încît să se poată executa la orice adresă (partiţie) din memoria internă. Acest lucru se poate rezolva

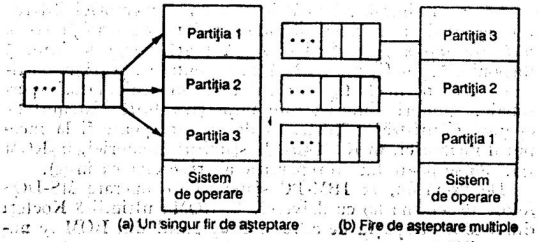
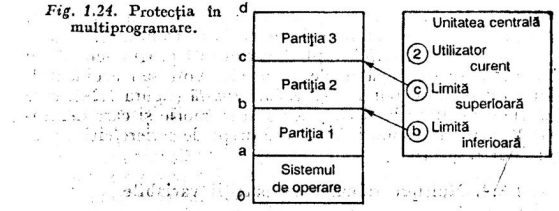


Fig. 1.23. Multiprogramare cu partiţii fixe.

Fig. 1.24. Protecţia în multiprogramare.



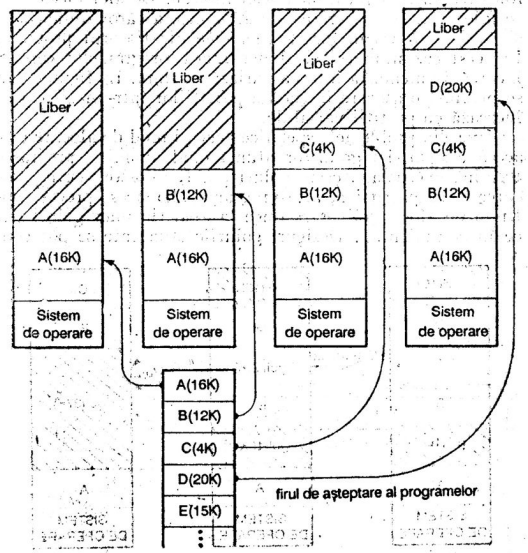


Fig. 1.25. Alocarea dinamică a partiţiilor.

fie prin modificarea programului în momentul încărcării, fie prin utilizarea unui registru de bază (relocare).

Problema *protecției* implică cerința ca programele să nu poată să se distrugă unele pe altele, voit sau accidental. În acest scop se pot folosi *registre limită* (figura 1.24), care să fie consultate la fiecare acces la memorie și care delimitează zona de memorie unde are drept de scriere/citire programul respectiv.

1.8.4. Multiprogramarea cu partiții variabîle

După cum s-a arătat, utilizarea partițiilor fixe duce la folosirea ineficientă a memoriei interne. De aici ideea de a se folosi *partiții variabile* sau *dinamice*, a căror dimensiune se adaptează cerințelor de memorie ale fiecărui program. În acest caz numărul și dimensiunea programelor care se încarcă în memoria internă variază în timp. În figura 1.25 se prezintă o alocare inițială a partițiilor într-un sistem ce lucrează cu partiții variabile.

Una din problemele specifice acestui mod de alocare este aceea a apariției golurilor atunci cînd anumite programe termină execuția. Aceste goluri pot fi folosite pentru alocarea unor partiții la alte programe, dar acest proces continuu de alocare/eliberare duce la apariția unor goluri din ce în ce mai mici. Desigur, golurile adiacente se pot con-

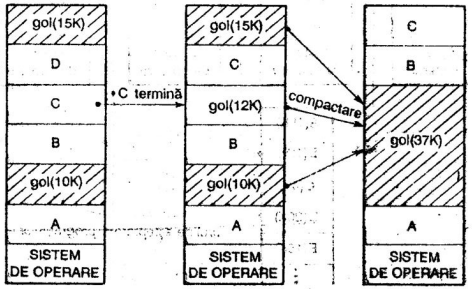


Fig. 1.26. **Compactarea golurilor.**

catena într-unul singur. Rămîne însă problema programării memoriei libere prin goluri neadiacente. De aceea în anumite momente este necesară adunarea tuturor acestor goluri într-unul mai mare — operațiune numită *compactare* sau *colectare goluri* (figura 1.26).

Operațiunea de compactare cere însă un important timp de unitate centrală; de aceea, ea nu trebuie făcută de către sistemul de operare la intervale prea scurte de timp. De altfel, multe sisteme de operare implementează compactarea doar ca o opțiune de generare care poate sau nu să fie selectată.

Una din problemele care trebuie rezolvate acum se referă la modul de a reține utilizarea memoriei interne. Vom propune în continuare 3 astfel de modele:

1.8.4.1. *Gestiunea memoriei prin harta de alocare*

Memoria internă este divizată în *unități de alocare*, cuprinse de obicei între cîțiva octeți și cîțiva Kocteți. Corespunzător fiecărei unități de alocare există un bit în harta de alocare, care este 0 dacă unitatea este liberă sau 1 dacă este ocupată (figura 1.27).

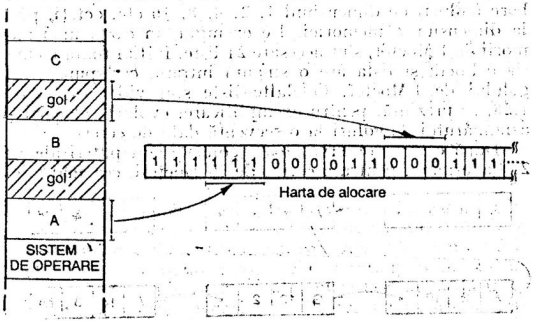


Fig. 1.27. **Hartă de alocare.**

Dimensiunea unității de alocare este foarte importantă. Dacă este mică, dimensiunea partițiilor este foarte apropiată de dimensiunea programelor, însă crește mult harta de alocare care ocupă cîteva procente din memoria internă. Dacă unitatea de alocare este mare, harta este mică, dar zone apreciabile de memorie rămîn nefolosite, deoarece fiecărui program i se alocă un număr întreg de unități. Alocarea la un proces a n unități presupune găsirea a n de 0 consecutivi în harta de alocare.

1.8.4.2. *Gestiunea memoriei prin liste înlănțuite*

În acest caz se întreține o listă înlănțuită cu segmentele de memorie (figura 1.28). Fiecare element al listei memorează dacă segmentul respectiv este alocat (A) este este gol (G), precum și dimensiunea și amplasamentul său. De obicei, lista este sortată crescător după adresele de memorie, lucru care face mai facile actualizarea și compactarea golurilor.

Lista poate fi simplu înlănțuită, capul ei fiind memorat într-o tabelă a sistemului de operare, fie dublu înlănțuită pentru a fi mai ușoară concatenarea grafurilor adiacente.

1.8.4.3. *Gestiunea memoriei prin fragmente*

Ideea constă în crearea unor liste separate a zonelor libere (goluri) de dimensiuni 1, 2, 4, 8, 16 etc. octeți, pînă la dimensiunea memoriei. De exemplu, în cazul unei memorii de 1 Moctet, sînt necesare 21 liste. Inițial toată memoria e liberă și lista are o singură intrare, corespunzătoare golului de 1 Moctet. Celelalte liste sînt vide. În figura 1.28, se prezintă, pentru exemplificare, evoluția alocării și a numărului de goluri la o secvență dată de cereri.

După cum se observă, toate golurile sînt puteri ale lui 2. Ca urmare, pentru o cerere de 70 Kocteți, cea mai mică

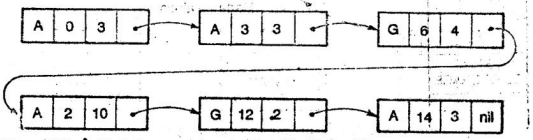


Fig. 1.28. Înlănțuirea zonelor de memorie pentru exemplul din fig. 1.27.

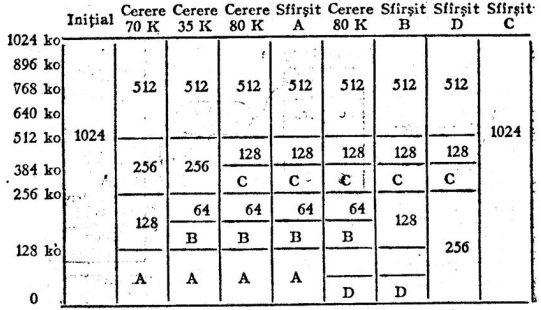


Fig. 1.29. Alocarea pe zone puteri de 2.

putere a lui 2 care cuprinde pe 0 are 128 Kocteți. Cum procedează sistemul de operare? „Sparge" blocul de 1 024 de Kocteți în două blocuri de 512 Kocteți, numite *fragmente*. Apoi unul din ele este divizat în 2 fragmente de 128 Kocteți. Dintre acestea, primul este alocat programului A. Apoi sosește cererea de 35 Kocteți. Se va rotunji în sus la 64 Kocteți. Sistemul de operare va diviza un fragment de 128 Kocteți în două de 64 Kocteți fiecare.

Să analizăm ce se întîmplă cînd un program eliberează o zonă de memorie (de exemplu, programul A). Zona de memorie alocată lui, de 128 Kocteți, se va atașa listei blocurilor libere, de 128 Kocteți. Cînd, de exemplu, apar 3 blocuri libere de 64 Kocteți la adrese consecutive, ultimele două vor forma prin fuziune un fragment mai mare, de 128 Kocteți.

Avantajul acestei metode de alocare este că dacă se eliberează un bloc de n octeți, sistemul v-a căuta doar în listele asociate acestei dimensiuni pentru eventuale compactări. Apare însă și un dezavantaj, acela al rotunjirii în sus al cererilor de multiplu de 2, ceea ce duce la o anumită fragmentare a memoriei și deci la utilizarea nu totdeauna eficientă a acestei resurse.

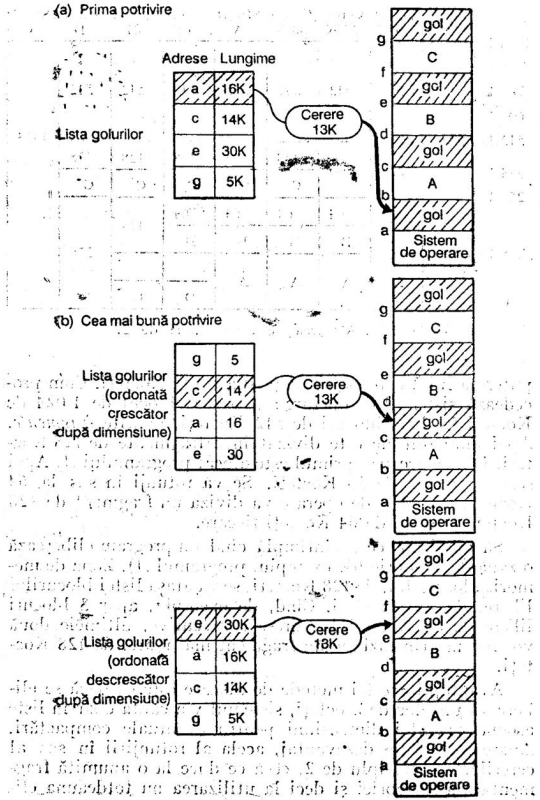


Fig. 1.30. **Strategii de alocare a memoriei în programe.**

1.8.4.4. *Strategii de alocare*

Strategiile de alocare sînt folosite pentru a determina în ce loc al memoriei interne se vor plasa programele. În figura 1.30 se prezintă trei algoritmi mai des folosiți. Este indicat să existe liste separate cu goluri și cu partițiile, iar lista de goluri să fie sortată după dimensiune.

(*a*) „Prima potrivire (*first-fit*) presupune căutarea în listă pînă cînd se găsește primul gol de dimensiune mai mare sau egală cu cererea programului. În cele mai multe cazuri, în urma alocării rămîne un mare gol, de dimensiune mai mică. O variantă a strategiei (*a*), „Următoarea potrivire" (*next-fit*), presupune, de asemenea găsirea primului gol suficient de mare, doar că procesul de căutare nu se declanșează la începutul listei, ci din poziția rămasă de la ultima căutare;

(*b*) „Cea mai bună potrivire" (*best-fit*) presupune căutarea în toată lista și alegerea acelui gol în dimensiunea cea mai apropiată de cerere a programului. Metoda este mai lentă, dar conduce la o mai bună utilizare a memoriei;

(*c*) „Cea mai proastă potrivire" (*worst-fit*) presupune căutarea în listă a golului cel mai mare, din care se alocă partiția pentru program. Golul rămas în urma alocării este mare și are șansa să fie suficient pentru satisfacerea altui program.

1.8.4.5. Multiprogramarea prin evacuare/reîncărcare

În toate schemele prezentate pînă acum programele odată încărcate, rămîneau în memoria internă pînă la terminare. O altă posibilitate o reprezintă și *evacuarea/încărcarea* programelor (*swapping*). Atunci cînd sosește o cerere de alocare și nu există nici un gol suficient pentru încărcarea programului, un program din memorie poate fi evacuat pe disc (*swapped out*) și astfel să se creeze spațiu pentru încărcarea programului apelant. După un timp,sistemul de operare va reîncărca (*swapped in*) programul întrerupt și îi va putea conferi controlul unității centrale.

Programele se află pe disc, de unde sînt încărcate și evacuate alternativ, în funcție de necesitățile de multiprogramare dintr-o zonă numită „swapping". În sistemele mari sofisticate, evacuarea se face doar atunci cînd ultima ima-

gine a programului a fost modificată prin execuție. În caz contrar, programul nu mai trebuie copiat pe disc, ultima lui imagine putînd fi folosită la următoarea încărcare. Se reduce astfel din timpul, destul de lung, de transfer al programelor între memorie și disc.

1.8.6. Memoria virtuală

Ideea de bază cu privire la conceptul de *memorie virtuală* constă în aceea că dimensiunea programelor, împreună cu datele aferente, poate depăși memoria fixă afectată (disponibilă). Sistemul de operare face ca numai partea curentă a procesului să fie prezentă în memoria internă, restul aflîndu-se pe disc. Acest concept nou, utilizat pentru prima dată la sistemul Atlas, realizat în anii 1960 la Universitatea Menchester din Anglia, este specific sistemelor de calcul multiprogramate.

Există două metode mai larg folosite pentru implementarea memoriei virtuale: *paginarea* și *segmentarea*.

Cheia conceptului de *memorie virtuală* constă în disocierea adreselor referite în cadrul proceselor în execuție de adresele reale disponibile în memoria internă. Adresele generate în program se numesc *adrese virtuale*, pe cînd cele disponibile în memoria internă sînt *adrese reale*. Programele care se execută adresează un *spațiu virtual*, V, care nu coincide cu *spațiul real* al calculatorului.

Spațiul virtual trebuie proiectat în spațiul real printr-un mecanism de *translatare dinamică a adreselor*. Toate aceste mecanisme au proprietatea că proiectează spațiul virtual contiguu într-unul real discontinuu, realizînd totuși o contiguitate artificială.

Deoarece spațiul virtual poate fi mare, cele mai multe sisteme cu memorie virtuală folosesc o *schemă cu 2 nivele de memorie*. Primul nivel îl constituie *memoria internă* (reală), în care se execută programele și unde se află fragmente ale acestora. Al doilea nivel îl constituie *memoria auxiliară*, în general pe disc, și unde se află memorate în întregime programele utilizatorilor. Deoarece memoria internă este utilizată simultan de mai multe programe care au spațiul virtual mult mai mare decît cel real, numai mici porțiuni din fiecare program se află în memoria internă la un moment dat.

1.8.6.1. *Paginarea*

Spațiul de adrese virtuale ale unui program este împărțit în *pagini virtuale*, care reprezintă porțiuni de dimensiune fixă ale programelor. Adresa virtuală într-un sistem paginat este reprezentată de perechea (p, d), unde p reprezintă numărul paginii în memoria virtuală (auxiliară), în care acesta este depus, iar d reprezintă deplasamentul (adresa relativă) în pagină. Dar fiecare program trebuie să se execute într-o memorie internă (reală). De aceea, paginile necesare trebuie mutate din memoria auxiliară în cea internă sub forma unor pagini reale, care au aceeași dimensiune cu cele virtuale (figura 1.31). Orice pagină poate fi plasată în oricare pagină reală disponibilă.

Translatarea dinamică a adreselor din spațiul virtual în cel real se face astfel: programul în execuție va utiliza adrese virtuale de forma $v = (p, d)$. Pe baza unei *tabele de pagini* se determină că paginii virtuale p îi corespunde pagina fizică, p . Adresa reală se va forma prin concatenarea deplasamentului d la pagina p (figura 1.32).

Există mai multe tehnici prin care se poate face proiectarea spațiului virtual în spațiul real. Vom prezenta trei dintre ele:

Translatare directă. În acest caz, tabela de pagini a fiecărui program conține cîte o intrare pentru fiecare pagină

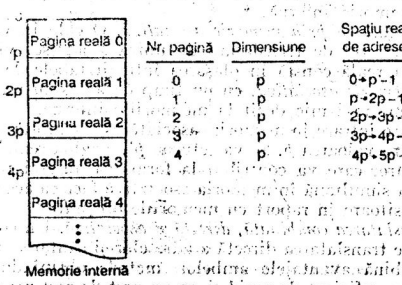


Fig. 1.31. Pagini reale în memoria internă.

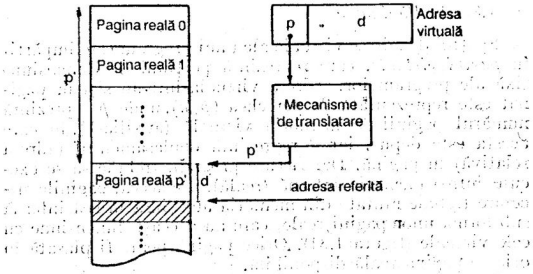


Fig. 1.32. Corespondența adresă virtuală — adresă reală.

din spațiul virtual. Aceste intrări asociază pagini reale paginilor virtuale și precizează dacă acestea sînt rezidente sau nu în memoria internă.

Adresa de bază a tabelei de pagini corespunzătoare acestui program, b, împreună cu numărul de pagini extras din partea superioară a adresei virtuale, p va genera o adresă în tabela de pagini. Aici se va vedea dacă pagina respectivă este rezidentă în memoria internă, iar în caz afirmativ, se va extrage de aici numărul real al paginii în memorie, p'. Se va genera în acest mod adresa reală cu care se poate apela direct memoria internă, r.

Translatarea prin memorie asociativă. O cale în vederea măririi vitezei de translatare dinamică a adresei virtuale în adresa reală constă în plasarea întregii tabele de pagini într-o *memorie asociativă*, cu un timp de acces cu un ordin de mărime mai mic decît la memoriile interne.

Fiecare intrare în memoria asociată este căutată simultan pentru valoarea p. Se va returna p', pagina reală corespunzătoare, care va contribui la formarea adresei reale, r. Căutarea simultană în memoria asociativă face ca aceasta să fie costisitoare în raport cu memoriile obișnuite.

Translatarea combinată, directă și asociativă. Un compromis între translatarea directă a adreselor și căutarea asociativă îmbină avantajele ambelor metode, obținîndu-se un mecanism suficient de rapid și cu un preț de cost rezonabil.

Programele în execuție se referă la adrese virtuale, $v = (p, d)$. Mecanismul de translatare caută mai întîi pagina p în memoria asociativă care conține o *tabelă de pagini parțială*, cu ultimile pagini folosite.

Dacă p este găsită aici, memoria asociativă returnează valoarea p', care concatenată la deplasamentul d formează adresa reală r. Dacă însă pagina p nu este găsită în memoria asociativă, se procedează la căutarea ei secvențială în tabela de pagini obișnuită.

Memoria asociativă nu trebuie să fie prea mare, pentru a nu fi prea scumpă. Practic, s-a demonstrat că utilizarea unei astfel de memorii formate din 8 sau 16 registre în completare cu translatarea directă conduce la un procent de 90% din performanțele de timp ale unui sistem pur asociativ.

1.8.6.2. *Segmentarea*

Paginarea furnizează o tehnică pentru implementarea unor spații liniare mari într-o memorie fizică limitată. Pentru o serie de aplicații însă este avantajoasă execuția lor într-o zonă de memorie reală contiguă.

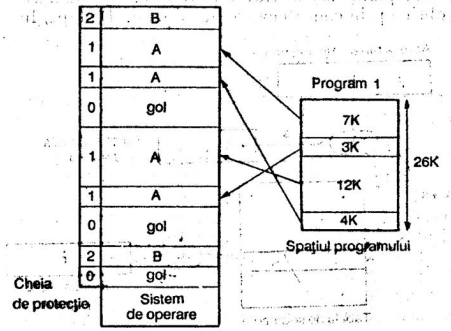


Fig. 1.33. Alocarea pe segmente necontigue.

În sistemele segmentate tocmai acest neajuns se înlătură, în sensul că devine posibil ca un program să ocupe mai multe blocuri de memorie, de dimensiuni diferite și care nu trebuie să fie adiacente.

Dar această metodă introduce și unele probleme noi. De exemplu, devine mult mai complexă protecția fiecărui program împotriva distrugerii din partea altor programe. O schemă posibilă este prezentată în figura 1.33. Se poate atașa fiecărui segment (bloc) o cheie de protecție care să fie distinctă de la un program la altul și gestionată de sistemul de operare.

Adresa virtuală într-un sistem segmentat, $v = (s, d)$, conține numărul de segmente, s, și deplasamentul în cadrul acestuia, d. Un program se poate executa numai dacă cel puțin un segment (cel curent) este prezent în memoria internă. Segmentele sînt transferate între memoria internă și disc ca niște unități distincte. Strategia de plasare a segmentelor în memorie este similară cu cea de alocare a partițiilor variabile.

În figura 1.34 se prezintă posibilitatea de translatare (directă) a adreselor virtuale în adrese reale. Se utilizează o *tabelă de segmente*, în care se determină ce segment real, s', îi corespunde segmentului virtual, s. Apoi adresa reală, r, se obține prin concatenarea lui s' cu d. De fapt, lucrurile

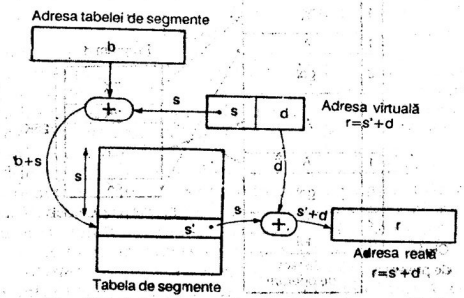


Fig. 1.34. Translatarea adreselor în cazul segmentării.

se desfășoară similar cu paginarea, doar că segmentele unui program sînt unități logice ale acestuia, definite și controlate de utilizator, spre deosebire de pagini care aveau dimensiuni fixe și erau invizibile programatorului. Relocarea fiecărui segment se face separat, în zone neadiacente, prin intermediul tabelei de segmente.

Desigur, sistemul de operare poate acorda fiecărui proces *dreptul de acces* distinct la diferitele segmente, care să asigure un control corect al accesului. Principalele funcții sînt:

— *citire* (R), ceea ce permite procesului să acceadă prin citire la orice zonă de informații din segment;

— *scriere* (W), ceea ce dă dreptul procesului să modifice zone ale segmentului respectiv;

— *execuție* (E), ceea ce permite procesului să lanseze în execuție segmentul respectiv;

— *adăugare* (A), ceea ce permite procesului să adauge informații adiționale la sfîrșitul unui segment, fără însă a modifica informația existentă.

Utilizînd cele 4 drepturi de acces prezentate, este posibilă crearea a 16 *moduri de control al accesului*, dintre care, desigur, numai unele au sens.

În modul 0 ($\overline{R}$, $\overline{W}$, $\overline{E}$), nu se permite nici un fel de acces la segment. Acest mod poate fi util în sisteme cu securitate ridicată, cînd nici un proces utilizator nu trebuie să aibă acces la anumite segmente. În modul 1 ($\overline{R}$, $\overline{W}$,E) este permisă doar execuția codului dintr-un segment. El este folosit atunci cînd se permite unui proces să utilizeze un program dintr-un anumit segment, fără însă a putea să-l copieze sau modifice.

În modul 4, de exemplu, ($\overline{R}$, $\overline{W}$, $\overline{E}$) se permite unui proces doar acces de tip „read-only", fără execuție sau modificare.

Schema simplă de protecție a segmentelor, prezentată mai sus, stă la baza multor mecanisme implementate în diverse sisteme de operare. În figura 1.35 se prezintă structura unei intrări în tabela de segmente. Se observă că, pe lîngă alte informații, aici se reține prin *bitul de rezidență*, r, dacă segmentul respectiv este sau nu în memoria internă. Dacă nu este prezent, se va genera o *întrerupere de segment*, în

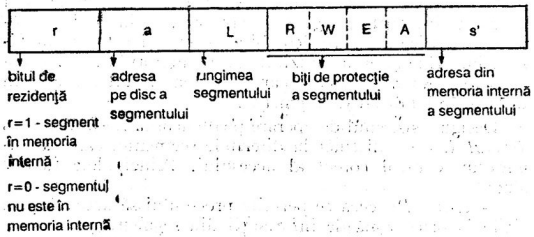


Fig. 1.35. Tabela de segmente.

urma căreia sistemul de operare va încărca segmentul apelat de pe disc în memoria internă. De asemenea, încercarea de a viola protecţia conferită segmentului va conduce la o *întrerupere de protecţie*.

1.8.6.3. *Segmentarea şi paginarea*

Atît segmentarea, cît şi paginarea oferă anumite avantaje, ce pot fi combinate într-un singur sistem care să folosească ambele concepte. Este şi soluţia adoptată în sistemele IBM 370 (TSS) sau Honeywell 6180 (Multics). Se pleacă de la premisa că segmentele au dimensiune multiplu al dimensiunii paginii şi ca urmare este inutil ca toate paginile unui segment să fie prezente simultan în memoria internă. În acest caz, adresa virtuală are un aspect tridimensional, $v=(s,p,d)$, unde s reprezintă numărul de segmente, p este numărul de pagini din cadrul segmentului, iar d reprezintă deplasamentul. Schema de translatare dinamică a adreselor este prezentă în figura 1.36.

Adresele folosite în program, sînt virtuale, $v=(s,p,d)$. O memorie asociativă reţine pentru paginile cele mai recent folosite (s,p) paginile reale asociate (p'), ceea ce contribuie la creşterea vitezei de acces la anumite pagini. În caz contrar, se parcurge schema clasică de translatare directă: cu numărul de segment s ce intră în tabela de segmente şi se

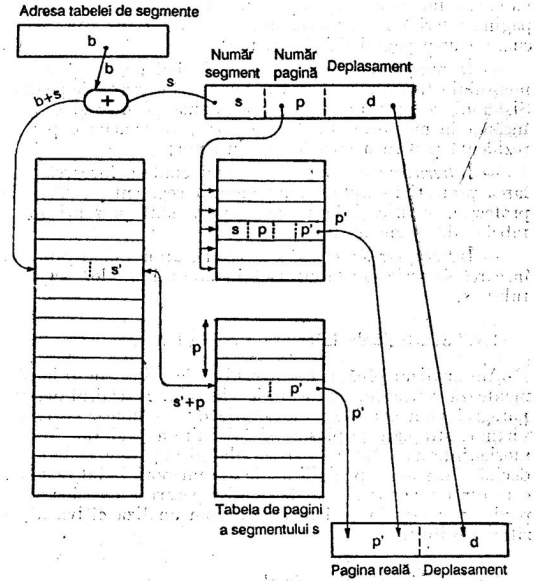


Fig. 1.36. Translatarea adreselor în sistemele cu segmentări şi paginare.

extrage adresa s' a tabelei de pagini a segmentului s. Apoi cu numărul de pagină p se va extrage adresa p' a paginii reale.

În acest proces de translatare se pot opera mai multe cauze care generează întreruperi:

— *Întreruperi de segment*, atunci cînd segmentul s nu nu este prezent în memoria internă. Sistemul de operare care va trata întreruperea, va localiza segmentul pe disc

va crea o tabelă de pagini pentru acel segment și va încărca pagina referită în memoria internă, după o eventuală evacuare a unei pagini deja existente;

— *Întreruperea de pagină*, atunci cînd pagina *p* a segmentului *s* (prezent în memorie) nu este încărcată de pe disc. Sistemul de operare va localiza pagina pe disc și o va încărca în memorie Dacă este cazul, va evacua o pagină rezidentă pentru a crea loc paginii noi;

— *Întrerupere de protecție*, atunci cînd se încearcă violarea protecției afectate unui anumit segment, *s*. Biții de protecție, verificați la fiecare acces, sînt prezenți în tabela de segmente;

— *Întrerupere de depășire segment*, atunci cînd procesul încearcă să scrie sau să citească în afara limitelor segmentului *s*.

1.8.7 Strategii de înlocuire a paginilor

Am analizat pînă acum mecanismele prin care se implementează conceptul de memorie virtuală. În cadrul acestui paragraf vom considera strategiile de gestiune a spațiului virtual: strategii de plasare și înlocuire a paginilor în memoria internă. Rutinele sistemului de operare trebuie să decidă dacă toate paginile reale ale memoriei sînt ocupate care din ele vor fi eliminate pe disc pentru a face loc altor pagini necesare în acel moment. Vom analiza cîțiva algoritmi posibili.

1.8.7.1 *Algoritmul optimal*

În acest caz se are în vedere faptul că principiul optimalității statuează că se obțin performanțe optime dacă pagina care va fi înlocuită nu va mai fi necesară în viitor sau, dacă o astfel de pagină nu există, se va înlocui aceea care va fi referită cel mai tîrziu. Desigur că o astfel de strategie nu poate fi implementată, deoarece nu pot fi obținute informații cu privire la evoluția ulterioară a programelor. Totuși, pentru a obține performanțe bune se poate aproxima acest principiu al optimalității, utilizînd unele prezumții rezultate din însăși logica celor mai multe programe.

1.8.7.2 *Algoritmul aleatoriu*

O cale situată la extremitatea opusă, este aceea de a nu avea în vedere nici un criteriu la alegerea paginii ce urmează să fie înlocuită. Decizia este luată aleator. Șansa tuturor paginilor de a fi selectate pentru înlocuire este egală. Desigur că poate fi aleasă chiar pagina care urmează să fie apelată. Acest mod de abordare este puțin folosit. În aceste două extreme se găsesc toți algoritmii utilizați în sistemele cu memorie virtuală. Figura 1.37 prezintă un set de curbe obținut prin simulare, care arată performanțele diferitelor metode de înlocuire a paginilor.

1.8.7.3 *Algoritmul FIFO*

În cazul algoritmului FIFO (*First in — first out*) se va asocia fiecărei pagini ora la care a fost încărcată în memoria internă. Cînd se cere înlocuirea unei pagini, se va alege aceea care se află în memorie de cel mai mult timp. Fondul intuitiv al algoritmului este simplu: paginile vechi în memoria internă cedează locul altora noi. Însă raționamentul nu este întotdeauna corect. Conform lui, s-ar putea elimina pagini foarte importante care, deși vechi, aparțin unor programe folosite în comun de mai multe procese.

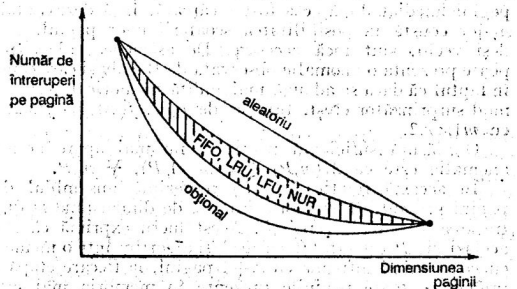


Fig. 1.37. Performanțele metodelor de înlocuire a paginilor

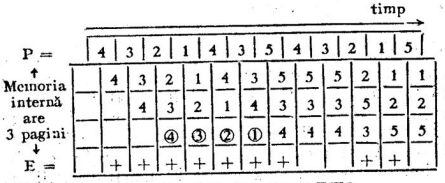


Fig. 1.38 Algoritmul FIFO

În figura 1.38 se prezintă modul de acțiune al acestui algoritm pe un exemplu dat. Pe primul rînd, într-o secvență arbitrară de pagini cerute, de cîte ori este nevoie de o pagină care nu este în memoria internă apare un eveniment numit „eșec". Cînd pagina apelată este prezentă în memoria internă are loc un „succes". Atunci cînd apare „eșec" și memoria este plină, algoritmul de eliminare va selecta pagina cea mai veche, aflată în vîrful stivei.

Se va defini o *funcție de succes*; $s=S/P$, unde S reprezintă numărul de evenimente „succes" la o secvență de P pagini, și o funcție de „eșec", $e=E/P$, unde E reprezintă numărul de evenimente „eșec" la aceeași secvență P de pagini.

Algoritmul are avantajul că evită eliminarea unei pagini imediat după ce a fost încărcată. Însă dezavantajul major constă în posibilitatea scoaterii unor pagini, care, deși vechi, sînt încă necesare. De asemenea, algoritmul poate prezenta o anomalie observată de Belady și care constă în faptul că dacă se adaugă mai multă memorie internă, în mod surprinzător crește funcția de eșec: $f(m1,p) < f(m2,p)$, cu $m1 < m2$.

O măsură suficientă pentru a nu mai apare această anomalie este ca $M(m,P) \subset M(m+1,P)$, $\forall\ m,P$.

În această relație $M(m,P)$ reprezintă ansamblul de pagini prezente în memoria internă de dimensiune m după tratarea șirului de cereri P. Acest lucru exprimă că dacă același șir P este tratat cu același algoritm într-o memorie cu m pagini și într-una cu $m+1$ pagini, la fiecare etapă de prelucrare toate paginile prezente în memoria mai mică sînt de asemenea prezente și în memoria mai mare.

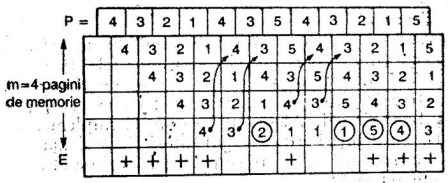


Fig. 1.39. Algoritmul LRU.

1.8.7.4 *Algoritmul LRU*

LRU (*Least recently used*) reprezintă unul din cei mai folosiți algoritmi de înlocuire a paginilor. El cere ca fiecare pagină să rețină timpul ultimei referiri, astfel încît să se poată selecta totdeauna pagina care nu a mai fost folosită de cel mai mult timp. Se pleacă de la premisa că o pagină referită de curînd are mai mari șanse de a fi folosită din nou. În figura 1.39 se prezintă evoluția algoritmului printr-un exemplu dat.

Pentru implementarea acestui algoritm se consumă încă importante resurse pentru memorarea timpului la fiecare acces la o pagină, precum și pentru sortarea paginilor în ordinea timpului de acces. De aceea, în realitate se implementează aproximări ale acestuia.

1.8.7.5 *Algoritmul LFU*

LFU (*Least frequently used*) reprezintă o aproximație a algoritmului LRU. Se asociază un contor software la fiecare pagină, inițial avînd valoarea 0. La fiecare întrerupere de ceas, sistemul de operare scanează toate paginile memoriei. La fiecare pagină bitul de referință, R (cu valoare 0 sua 1 pusă hard la referirea paginii), este adunat la vechea valoare a contorului. În acest mod contoarele indică, cu oarecare aproximație, de cîte ori a fost referită o pagină.

Algoritmul de eliminare va alege pagina care are contorul cu valoarea cea mai mică.

1.8.7.6 *Algoritmul NUR*

NUR (*Not used recently*) reprezintă unul din algoritmii cei mai folosiți, fiind și el o aproximare a lui LRU. Paginile care nu au fost folosite în ultimul interval de timp au șansa de a fi refolosite și în continuare; ca urmare, pot face obiectul eliminării. Algoritmul poate fi implementat în conjuncție eu 2 biți hard corespunzători fiecărei pagini:

— *bitul de referință* (R), care se pune pe 1 ori de cîte ori o pagină este referită;

— bitul de modificare (M), care este poziționat pe 1 atunci cînd pagina a fost modificată pentru scriere.

Algoritmul lucrează astfel: inițial, biții de referință ai tuturor paginilor sînt zero. Cînd apare o referință la o anumită pagină, bitul R corespunzător se pune pe 1. La fel, inițial toți biții de modificare, M, sînt pe zero. Atunci cînd se face scrierea la o locație din pagină, bitul M corespunzător este selectat la 1. Cînd trebuie înlocuită o pagină bitul M corespunzător este selectat la 1. Cînd trebuie întocmită o pagină care nu a fost referită, $R=0$. După găsirea unei astfel de pagini se verifică dacă a fost modificată ($M=1$), caz în care eliminarea presupune scrierea ei înapoi pe disc. Dacă $M=0$ se face pur și simplu acoperirea vechii pagini cu cea nouă.

1.8.7.7. Dimensiunea paginilor

În centrul strategiilor de gestiune a memoriei se află conceptul de *localitate*, conform căruia procesele tind să refere locațiile memoriei în mod neuniform, cu precădere în anumite intervale de timp.

Se poate vorbi de două tipuri de localități. *Localitatea temporară* se bazează pe observația că dacă o dată sau instrucțiune este referită, atunci există posibilitatea ca în intervalul de timp următor ea să fie referită din nou. *Localitatea spațială* implică faptul că dacă o locație a fost referită, există o posibilitate mare ca o adresă vecină să fie apelată.

Peter Denning a enunțat astfel *principiul localității*:

1) În timpul unui interval de timp un program face referiri la paginile sale în mod neuniform;

2) Frecvența cu care se face referire la o anumită pagină este o mărime cvasistaționară, adică tinde să-și schimbe valoarea lent.

Principiul corespunde unei realități, deoarece programatorii folosesc ciclurile și secvențele liniare ca elemente de structură de baza ale programelor. Ele rămîn într-o zonă de program, numită localitate, un timp mai îndelungat. Efortul de evacuare/încărcare a paginilor va fi concentrat la salturile de la o localitate la alta, cînd factorul de corelare și previziune a saltului la paginile necesare este aproape nul.

Localitatea temporară se bazează pe utilizarea în cadrul programelor a unor cicluri, subrutine, stive, variabile contor. Localitatea spațială are ca suport folosirea în progresie a matricelor, a secvențelor de cod și a tendințelor programatorilor de a pune unele lîngă altele definițiile de variabile.

Tot Denning dezvoltă o nouă viziune asupra modului de utilizare a paginilor, introducînd conceptul de *set de lucru*. Acesta reprezintă o colecție de pagini pe care un proces le referă în mod activ. Pentru ca un program să se execute în mod eficient, paginile setului-său de lucru trebuie să fie prezente în memoria internă.

Setul de pagini de lucru ale unui proces, $W(t, w)$, la timpul t, este format din paginile care au fost referite în intervalul de timp t-w, t (vezi figura 1.40).

Variabila w, numită *fereastra setului de lucru*, este foarte importantă, iar dimensiunea ei influențează în mod esențial performanțele sistemului

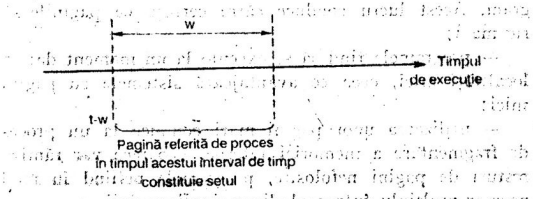


Fig. 1.40. **Definiția setului de pagini de lucru.**

Setul de lucru se schimbă pe măsură ce un proces se execută. Unele pagini sînt adăugate, altele eliminate. Probleme dificile apar atunci cînd un proces intră într-o zonă a execuției care schimbă total setul de lucru (practic, execuția se mută dintr-o localitate în alta).

Desigur, utilizarea acestui concept în gestiunea memoriei ridică o serie de dificultăți legate de estimarea setului de lucru necesar în momentele următoare, ținînd cont că nu există informații asupra evoluției ulterioare a proceselor.

Una din problemele esențiale în sistemele paginate de gestiune a memoriei o constituie *dimensiunea paginii*. Mărimea acestora influențează în mod hotărîtor performanțele sistemului. De aceea, pentru a vedea dacă ele trebuie să fie mari sau mici, să facem cîteva considerații:

— paginile mici duc la o utilizare bună a memoriei, însă fac ca dimensiunea tabelei de pagini să crească. În sistemele în care tabelele de pagini sînt în memoria internă este nevoie ca paginile să fie mari, astfel încît tabela să ocupe cît mai puțin loc;

— paginile mari, ca o parte din informația încărcată în memorie o dată cu paginile, trebuie să nu fie referite și deci să nu ocupe inutil memoria;

— deoarece operațiunile de I/E cu discul sînt relativ mari consumatoare de timp, trebuie minimizat pe cît posibil numărul de transferuri în timpul execuției unui program. Acest lucru conduce către cerința ca paginile să fie mari;

— programele tind să se execute la un moment dat în localități mici, ceea ce avantajează sistemele cu pagini mici;

— utilizarea unor pagini mari conduce la un proces de fragmentare a memoriei interne, deoarece vor rămîne resturi de pagini nefolosite, programele nefiind în mod necesar multiplu întreg al dimensiunii paginii.

O serie de studii teoretice și observații empirice conduce către ideea că cel mai bine este ca dimensiunea paginilor să fie mică. Iată cîteva din dimensiunile de pagini la unele dintre cele mai folosite sisteme de calcul: Honeywell — — 1 024 octeți; IBM-360 — 1 024 octeți; IBM-370 — 1 024 octeți sau 512 octeți; PDP 20 — 512 octeți; VAX 11/780 — 128 octeți.

Stabilirea dimensiunii paginii reprezintă o opțiune foarte importantă și trebuie să aibă la bază serioase studii și experimente privind modul de folosire a memoriei interne de către programe.

Capitolul 2

SISTEMUL DE OPERARE UNIX

Sistemul de operare UNIX a devenit rapid unul din cele mai populare sisteme de operare din lume. Popularitatea sa unică este în principal rezultatul portabilității acestui sistem de operare: UNIX este disponibil pe o gamă largă de calculatoare, în timp ce toate celelalte sisteme de operare sînt dedicate unui anumit calculator. Din această rațiune, dar și din altele, UNIX a devenit universal, tot așa cum a fost FORTRAN-ul limbajul universal o bună perioadă de timp. Și tot așa cum FORTRAN-ul a influențat stilul altor limbaje de programare, caracteristicile UNIX-ului au început să fie vizibile în multe alte sisteme de operare realizate în ultimul deceniu.

2.1. SCURT ISTORIC

Prima versiune de UNIX a fost elaborată în 1969 de Ken Thompson și Denis Ritchie în grupa de cercetare de la Bell Laboratories, pentru a fi folosită pe un calculator PDP-7. Ritchie a lucrat anterior la sistemul de operare MULTIX, care a avut o puternică influență asupra noului sistem de operare. În continuare, Ritchie și Thomson au lucrat la versiunea a doua a UNIX-ului pentru PDP 11/20.

A treia versiune, încă necomercială, este scoasă de cei doi cercetători în 1973 prin rescrierea completă în limbaj C pe PDP 11/45 și 11/70. Acum se adaugă importante facilități, printre care și cele legate de multiprogramare.

Prima versiune comercializată de Bell Labs a fost V6, în 1976.

În anul 1978 este distribuită versiunea V7, care putea lucra pe calculatoare PDP 11/70, Interdata 8/2 și VAX-minicalculatoarele de 32 de biți ale firmei DEC (versiunea 32V). Această versiune a fost distribuită gratuit în toate universitățile din S.U.A. Distribuirea în cod sursă a dus la creșterea popularității acestui sistem în mediile universitare, noile generații de utilizatori fiind pregătite în spiritul UNIX.

În această perioadă se creează la AT&T grupul USG (*U*nix *S*upport *G*roup), care își asumă controlul și responsabilitatea de distribuire a UNIX-ului, ce devine un produs, nu numai un obiect de cercetare. Grupul de la Bell Labs a continuat cercetările proprii, elaborînd în 1985 versiunea V8, disponibilă însă numai în interiorul laboratoarelor.

Prima versiune distribuită de USG a fost System III, în 1982.

Ea a încorporat facilitățile din V7 și 32V. În 1983, USG lansează noua versiune de UNIX, System V. După reorganizarea USG în USDL (*U*nix *S*ystem *D*evelopment *L*aboratory) se livrează, începînd cu 1984, System V-Release 2(V2).

Dimensiunea relativ mică, modularitatea și claritatea proiectării UNIX-ului au declanșat cercetări similare la numeroase alte firme (Rand, DEC) sau universități (Illinois, Harvard).

Cea mai puternică influență dintre sistemele UNIX care nu aparțineau lui Bell Labs sau AT&T au avut-o cele create la Universitatea California din Berkely.

Aici s-au realizat versiunile BSD-3 și BSD-4, care reprezintă îmbunătățiri ale versiunii 32V (BSD — *B*erkely *S*oftware *D*istributions). Astfel, în BSD 4.1 se introduc facilități de gestiune a memoriei virtuale prin paginare la cerere, iar în BSD 4.2 facilități de lucru în rețele locale (de tip Ethernet) sau rețele extinse (de tip Arpanet).

În plus, Berkeley a adoptat o serie de facilități de la noile sisteme de operare, pentru a crește puterea UNIX-ului. Au fost create o nouă interfață utilizator (C-Shell), noi editoare de texte (VI și EX), noi compilatoare pentru PASCAL și LISP etc.

BSD 4.2 poate fi comparată ca eficiență cu sistemul VMS de pe VAX.

Versiunile BSD-3 și BSD-4 au fost create pentru calculatoare VAX, iar BSD-2 pentru PDP 11. Noutățile aduse de implementările UNIX SYSTEM V și UNIX BSD se referă la:

— ameliorarea gestiunii fizice a fișierelor;

— dezvoltarea bibliotecii de rutine sistem (mecanisme de comunicare interprocese);

— dezvoltarea programelor utilitare.

În fig. 2.1 se reprezintă relația dintre diferitele versiuni ale sistemului UNIX pînă la nivelul anului 1984.

Din cauza atitudinii diferite a producătorilor de hard față de fenomenul UNIX, în care noii sosiți vedeau în UNIX o sursă bogată de soft fiabil. AT&T încearcă să impună o versiune standard, ca sigură alternativă de evoluție a propriului sistem.

În 1985, AT&T elaborează prima versiune a normelor SVID (*System V Interface Definition*).

Crearea în 1988 (la numai o lună după anunțarea de AT&T a versiunii 4 a sistemului UNIX System V) de către marii producători IBM, Hewlett-Packard, DEC, Hitachi, Philips a societății *Open Software Fundation* (OSF) reprezintă o acțiune de contracarare a atitudinii de monopolizare a AT&T-ului.

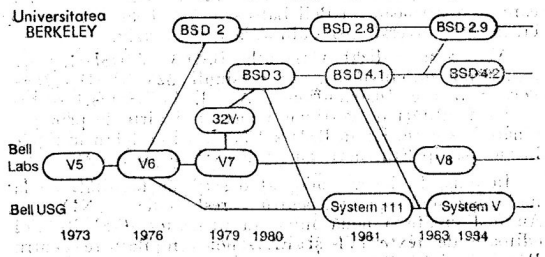


Fig. 2.1. Evoluția versiunilor UNIX.

Scopul declarat al societății OSF este de a dezvolta un sistem UNIX într-adevăr standard și deschis, pornind de la sistemul AIX care este bazat pe UNIX System V (versiunea 2) al IBM.

Aceste eforturi de standardizare au loc în contextul intensificării activităților de același gen ale altor organisme internaționale. Astfel, puternica organizație internațională IEEE (The *I*nstitute of *E*lectrical and *E*lectronics *E*ngineers Inc.) definește normele POSIX (*P*ortable *O*perating *S*ystem *I*nterface for Computing Environment). Acest standard, cu unele excepții, este identic cu normele SVID ale AT&T.

Un alt organism internațional care va avea un rol important în definirea standardelor relative la UNIX, este organizația X/Open. Scopul declarat al organizației este de a dezvolta sisteme deschise (independente de producător) care să devină standarde multi-vînzător. Astfel, organizația apără interesele utilizatorului, eliberîndu-l de dependența de un anume echipament.

Viitorul sistemului UNIX va depinde în principal de cele patru orientări de standardizare OSF, AT&T, POSIX, X/Open, însă în mare măsură el va fi decis de lupta de influență a celor două tabere: OSF, pe de o parte, și AT&T, pe de alta.

În disputa sa cu OSF, AT&T are un atu important de natură economică, și anume încheierea cu forțele aeriene ale armatei americane a unuia din contractele secolului, în valoare de 929 milioane dolari. O clauză contractuală prevede că toate mașinile beneficiarului să funcționeze sub sistemul de operare UNIX System V care să fie compatibil cu toate sistemele de operare UNIX eterogene, existente în armata americană.

Setul curent de sisteme de operare UNIX nu se limitează doar la Versiunea 8, System V (V.2) și BSD 4.2. S-au creat, datorită popularității acestui sistem, alte versiuni pe diferite calculatoare, unele sub licența UNIX, altele care doar asigură o funcționalitate de tip UNIX. Dintre acestea menționăm doar pe cele mai importante:

— ULTRIX 11 și ULTRIX 32, realizată de DEC pentru calculatoarele sale PDP 11 și respectiv VAX, sub licența V7, respectiv BSD 4.2;

— XENIX, licența V 7 System III, realizat de Micro soft pe calculatoarele personale IBM PC/XT;

— XENIX IBM, realizat de Microsoft și IBM pentru IBM PC/AT;

— AIX elaborat de IBM, care încearcă să supraliciteze facilitățile UNIX-ului în speranța impunerii unui standard;

— A/UX al firmei Apple;

— VENIX, realizat de Ventur pentru IBM PC/XT;

— ZEUS, realizat de Zilog pentru calculatoare System 8000;

— HP-XX, realizat de Hewlett-Packard pentru HP-Integral.

Firma AT&T s-a angajat să suporte implementarea sistemului UNIX pe microprocesoare, ca Intel IAPX-286, Motorola 68000, National 32032, ceea ce a sporit aria de utilizare a sistemului.

Realizarea în țara noastră a unei tehnici de calcul variate în domeniul mini și micro, necesitatea asigurării unei standardizări din punctul de vedere al interfeței utilizator și a unei portabilități superioare a produselor program au condus la inițierea în 1984, în cadrul ITCI, a proiectului sistemului de operare U, compatibil cu UNIX V7. S-a realizat apoi o altă versiune compatibilă cu System V (Release 2). Acest proiect vizează realizarea unui sistem de operare U atît pe minicalculatoarele din familia Independent și Coral, cît și pe microcalculatoare de 16 biți (M 216, Felix PC).

2.2. CARACTERISTICI ALE SISTEMULUI UNIX

UNIX este un sistem de operare cu destinație generală, multiutilizator, multiproces și cu regim de lucru interactiv. La fel ca RSX-11, UNIX lucrează în time-sharring. Să încercăm să subliniem care sînt cauzele deosebitei sale popularități:

1. UNIX este un sistem de operare atractiv pentru utilizatori. Are o interfață relativ simplă, un mare număr de programe utilitare, care acopereau o gamă largă de aplicații (baze de date, inteligență artificială, rețele, grafică, instruire asistată de calculator etc.). Sub UNIX sînt implementate

multe limbaje de nivel înalt și medii de programare: BASIC, FORTRAN 77, PASCAL, C, COBOL, LISP, PROLOG etc.;

2. UNIX este un sistem multiutilizator, permițînd lucrul simultan la mai multe terminale și deci o programare pe scară largă;

3. UNIX este un sistem portabil, fiind scris în cea mai mare parte în limbaj C. Acest aspect favorizează implementarea lui pe diferite calculatoare cu un preț relativ scăzut;

4. UNIX este larg folosit în universități, ceea ce a făcut ca mii de absolvenți să contribuie la răspîndirea lui în toată lumea.

Dintre caracteristicile mai importante ale sistemului UNIX amintim:

— sistemul UNIX este în clasa sistemelor de tip timesharring. El este multiutilizator, multiproces și interactiv;

— UNIX are un sistem de fișiere ierarhizat, arborescent, cu o organizare simplă a fișierelor, grupate în directoare;

— UNIX dispune de un sistem de intrare/ieșire generalizat, care tratează identic fișiere, periferice, memoria și comunicațiile interprocese;

— dispune de o interfață utilizator simplă, numită Shell, care însă se poate extinde la nivelul unui limbaj de programare puternic, de tip C, cu un interpretor independent de nucleu;

— UNIX are implementat un mecanism simplu de gestiune a memoriei, care asigură schimbul de pagini între memoria internă și disc (memoria virtuală), asigurînd un nivel de multiprogramare ridicat;

— UNIX asigura protecția informațiilor (fișiere și parole) și a mediului de execuție. Protecția între utilizatori se asigură prin parolă, cunoscută doar de proprietarul ei și codificată cu un mecanism complex de cifrare. În plus, fiecărui fișier i se asociază 9 biți de protecție care stabilesc drepturile de acces ale diferitelor clase de utilizatori;

— procesele sînt reentrante, adică toate procesele care folosesc același program utilizează o copie unică rezidentă în memorie.

2.3. STRUCTURA SISTEMULUI UNIX

Sistemul UNIX conține un nucleu (partea rezidentă) și un mare număr de utilitare accesibile prin intermediul interpretorului de comenzi Shell. Interfața tuturor acestor utilitare (ca și a programelor utilizator) cu nucleul se face prin intermediul unui ansamblu de funcții sistem. Utilizatorul are trei niveluri de acces la sistem:

— prin utilitare (nivelul extranucleu);
— prin funcții de bibliotecă standard a limbajului C (nivel intermediar);
— prin directive sistem (nivel scăzut).

În fig. 2.2 se prezintă schematic structura sistemului UNIX.

Se observă că nucleul este intermediarul între interfața furnizată de apelurile sistem și echipamentele fizice. Nucleul realizează gestiunea fișierelor, a memoriei, planificarea unității centrale între procese.

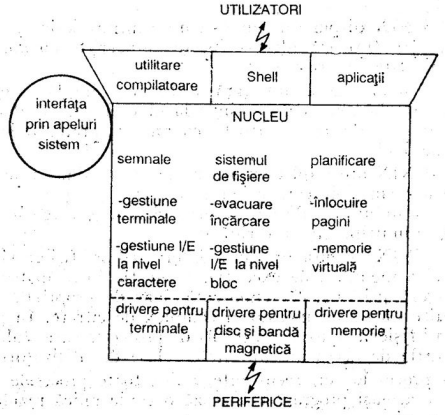


Fig. 2.2. Structura sistemului UNIX.

Apelurile sistem definesc interfața programator.

Apelurile sistem pot fi grupate în trei categorii:

— pentru manipularea fișierelor (perifericelor);
— pentru controlul execuției proceselor;
— pentru manipularea informației.

2.4. SISTEMUL DE FIȘIERE

În accepția UNIX-ului, un fișier reprezintă o succesiune de octeți. Fișierele sînt organizate într-o structură de arbore formată din directoare. Acestea sînt și ele fișiere care conțin informații de regăsire a altor fișiere.

Ele permit o grupare a fișierelor din punct de vedere logic în directoare. Sistemul permite crearea atîtor directoare, cîte sînt necesare. A da posibilitatea să se creeze directoare este un aspect pozitiv, dar care nu rezolvă integral problema. Ideea care stă la baza sistemului de fișiere la UNIX este ierarhia.

Fiecare director poate conține alte directoare, care la rîndul lor conțin alte directoare, pe un număr de niveluri nelimitat.

Acest lucru permite o grupare optimă a fișierelor în directoare.

Un alt aspect rezolvat în condiții foarte bune la UNIX îl constituie sistemul de protecție al fișierelor.

Există un control riguros al accesului la fișiere (periferice) absolut necesar în sistemele multiutilizator. Una din noile facilități ale UNIX este asocierea unor fișiere speciale fiecărui dispozitiv de I/E, aspect asupra căruia vom reveni.

În UNIX se deosebesc trei tipuri de fișiere: fișiere ordinare, fișiere director, fișiere speciale.

2.4.1. Fișiere ordinare

Un fișier ordinar este folosit pentru memorare de informații pe suport magnetic; el poate conține un program sursă (text) sau unul sub formă de cod executabil (binar), precum și date necesare execuției programelor.

Fişierele ordinare reprezintă singurul tip de fişiere folosite pentru memorarea pe termen lung a informaţiilor.

Numele fişierelor poate fi de maximum 14 caractere (tipăribile) şi nu pot exista în acelaşi director mai multe fişiere cu acelaşi nume. Este posibil ca acelaşi fişier să poarte diferite nume (vezi comanda **ln**). Deoarece în orice director sistemul foloseşte cu destinaţie specială numele '.'şi'..' acestea nu pot fi folosite de utilizatori.

Sistemul UNIX nu impune anumite convenţii cu privire la numele fişierelor. Cu toate acestea, anumite programe sistem cer ca fişierele pe care le prelucrează să poarte anumite sufixe: **.sh**, pentru programe Shell, **.bas**, pentru programe BASIC, **.c**, pentru programe C.

Fişierele care conţin cod executabil nu au de obicei sufix.

Există două tipuri de fişiere ordinare: **text** şi **binare.**

Fişierele text conţin numai caractere ASCII, pe cînd cele binare conţin toate cele 256 de valori posibile pentru fiecare octet.

Din punctul de vedere al sistemului de operare, în fişier există un şir de octeţi de lungime nedefinită, terminat cu un caracter special, EOF. Din punct de vedere utilizator, fişierele text se subdivid în linii terminate cu NL.

În general, sistemul nu recunoaşte nici o structură logică a informaţiei din fişier, acest lucru revenind în exclusivitate programului utilizator.

2.4.2. Fişiere director

Directoarele sînt fişiere care conţin liste (cataloage) de fişiere. Sistemul de directoare este întreţinut de sistemul de operare. Programele pot citi directoarele, dar sistemul are grijă ca acestea să nu modifice directoarele, pentru a se putea garanta integritatea sistemului de directoare. Toate operaţiile de adăugare a unui fişier nou în director sau de ştergere fişiere se fac de programe prin apelul unor funcţii sistem.

Un director este organizat ca o tabelă de traducere nume-identificator intern (i-nod). Fiecare utilizator are un

director propriu, în care,îşi depune propriile fişiere. Într-un director se ataşează fişiere ordinare sau alte directoare care grupează colecţii de fişiere cu aceeaşi destinaţie.

Sistemul de operare nu lucrează în interior cu numele fişierului, ci cu un număr numit i-nod. În orice fişier director există şi fişierele cu numele . care identifică i-nodul propriu al directorului şi .. ce identifică i-nodul directorului „tată" în care acesta apare ca subdirector.

În timpul unei sesiuni de lucru utilizatorul se poate muta dintr-un director în altul,care devine astfel director curent, cu ajutorul comenzii **cd**. Se observă că fişierele sînt grupate în directoare, iar directoarele sînt organizate într-o ierarhie.

Se creează astfel imaginea unui sistem de fişiere ierarhizat sub forma unui arbore, în vîrful căruia se află **directorul rădăcină-root** (notat /). O ierarhie tipică, dar mult simplificată, este prezentată în fig. 2.3.

După cum se poate observa, în vîrful piramidei se află directorul **root**, la care se pot conecta un număr de directoare; acestea, la rîndul lor, conectează directoare sau fişiere. Structura indicată este tipică pentru toate sistemele

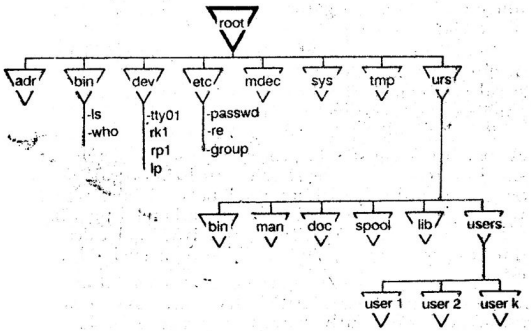


Fig. 2.3. **Structura standard a sistemului de fişiere.**

UNIX, principalele directoare folosite de sistem fiind următoarele:

/bin — care conține repertoarul utilitarelor uzuale (format executabil);

/dev — conține repertoarul fișierelor speciale;

/etc — conține repertoarul unor utilitare speciale de întreținere, folosite doar de administrator, fișierul de parole, de grupuri etc;

/tmp — conține fișiere temporare folosite de editor, compilatoare;

/lib — conține biblioteca de limbaje (C, FORTRAN etc.);

/usr — este cel mai mare director din sistem. În el se găsesc mai multe subdirectoare;

bin: cu utilitare mai rar folosite;

man & doc: cu manuale și documentație;

spool: cu fișiere pentru imprimantă și poștă electronică;

users: la care se conectează utilizatorii.

Disciplina parcurgerii arborelui implică restricția ca un director să nu fie legat în sus decît într-un singur loc (directorul „tată"), legătura cu acesta fiind făcută prin intrarea .. din director.

Pot exista mai multe fișiere cu același nume, dar plasate în directoare diferite; ca urmare sistemul nu identifică fișierul prin numele său, ci printr-o cale de acces, care reprezintă poziția acestuia în arbore.

Fișierele din directorul curent sînt direct accesibile; ele pot fi referite simplu prin numele lor. Fișierele care nu sînt în directorul curent se referă utilizînd o cale de acces. Se poate folosi o **specificare absolută,** cînd calea pleacă din root (/), indicîndu-se apoi nivelurile arborelui pînă la fișierul desemnat, fie o **specificare relativă,** cînd calea pleacă din directorul curent. Sintaxa generală a unei căi absolute este:

/dir1/dir2 ... /dirn/fișier,

iar a unei căi relative:

diri/diri + 1/ ... /diri + m/fișier.

Cîteva reguli simple se aplică la toate căile de acces:

(1) Cînd calea începe cu „/", specificarea pornește din directorul rădăcină (root), iar celelalte căi pornesc din directorul curent;

(2) O cale este o listă de nume, separate prin „/", sau un singur nume. Numele inițiale în liste sînt directoare, iar ultimul nume este fișierul desemnat;

(3) Se poate urca în arborele sistemului de fișiere prin specificarea numelui „...". în cale. Toate celelalte nume din cale coboară în arbore. Pentru exemplificare, să considerăm arborele indicat în fig. 2.3. Fișierul **date** din directorul **bin** se poate specifica:

/bin/date, printr-o cale absolută sau

../../../bin/date, printr-o cale relativă,

ce pleacă, de exemplu, din directorul **users**.

Specificarea corectă a căilor de acces reprezintă unul din aspectele esențiale în utilizarea sistemului de operare UNIX. Să remarcăm însă că pentru a accesa un fișier este de multe ori preferabil să schimbăm directorul curent (cu comanda **cd**) decît să indicăm repetat căile lungi.

2.4.3. Fișiere speciale

În sistemul UNIX toate perifericele sînt tratate ca fișiere, fiind denumite **fișiere speciale.** În acest mod fiecare dispozitiv fizic de I/E este identificat cu un fișier special.

Din punctul de vedere al utilizatorului nu există nici o diferență între lucrul cu un fișier ordinar și un fișier special.

Numele fișierelor speciale este indicat în directorul /dev; vom găsi, de exemplu:

— „**/dev/lp**" (fișierul special asociat imprimantei),

— „**/dev/tty2**" (fișierul special asociat discurilor de masă) etc.

Tratarea perifericelor ca fișiere speciale are o serie de avantaje, dintre care amintim;

— fișierele speciale sînt supuse unui mecanism de protecție similar cu cel al celorlalte fișiere;

— în vederea schimbului de date cu perifericele se pot folosi aceleași comenzi (utilitare) ca și în cazul fișierelor obișnuite, ceea ce conferă eleganță și simplitate în utilizare;

— utilizatorii pot efectua schimburi de date la/de la periferice fără să fie nevoiți să știe detalii din mecanismul lor de funcționare. O serie de periferice, ca de exemplu terminalele, schimbă cu calculatorul cîte un caracter la un moment dat.

Fișierele speciale care furnizează legături cu astfel de periferice se numesc fișiere speciale de tip caracter.

Aceste dispozitive, precum discurile sau benzile magnetice, transferă la/de la calculator grupe (blocuri) de mai multe caractere (de exemplu 512); fișierele speciale asociate lor se numesc fișiere speciale de tip bloc.

Deoarece nu există caractere memorate într-un fișier special, cîmpul lungime de fișier dintr-o listare a caracteristicilor fișierului (cu comanda **ls**) va conține în loc de lungimea fișierului, două numere:

— numărul major al perifericului, care indică tipul dispozitivului;

— numărul minor al perifericului, care specifică care din dispozitivele de același tip (dacă există mai multe în configurație) este asociat cu acest fișier special.

2.4.4. Mecanismul de protecție a fișierelor

Mecanismul de protecție este destinat controlului accesului la fișiere. Schema implementată în UNIX este simplă, realizînd în același timp o foarte bună separare a contextelor de execuție.

Există trei tipuri de acces la un fișier: read (r), write (w), și execute (x). Dreptul de citire (r) permite vizualizarea conținutului unui fișier; dreptul de scriere (w) permite modificarea fișierului, dreptul de execuție (x) permite încărcarea fișierului în memorie și lansarea programului cod obiect, reprezentat de conținutul acelui fișier sau citirea și execuția unui fișier de comenzi Shell.

În cazul directoarelor drepturile r, w, x sînt interpretate altfel.

Dreptul de citire (r) pentru un director permite utilizatorului să deschidă și să citească fișierul director cu comanda (**ls**).

Dreptul de scriere (w) implică posibilitatea creării sau ștergerii de fișiere într-un director. În fine, dreptul de execuție (x), semnifică faptul că este permis ca sistemul să caute în director în cursul prelucrării unei căi de acces.

Dacă un fișier este specificat printr-o cale, se caută în fiecare director al căii pentru determinarea următorului nume de secvență indicată. Interzicerea dreptului *x* pentru un director reprezintă o protecție importantă împotriva utilizatorilor care vor să folosească fișierul din acest catalog.

Există trei categorii de utilizatori: proprietar (**u**), grup (**g**) și ceilalți utilizatori (**o**). Fiecare fișier are un **proprietar** de obicei în persoana celui care le-a creat. Mai mulți utilizatori, care fac de obicei parte din același compartiment de lucru, formează un **grup**. Sistemul de protecție a accesului la fișier se bazează pe confruntarea cererilor utilizatorului (r, w, x) cu drepturile afectate categoriei din care face parte acesta (u, g, o). Pentru precizarea completă a drepturilor de acces la un fișier sînt necesari 9 biți: 3 indică drepturile proprietarului, 3 indică drepturile grupului, iar 3 indică drepturile celorlalți utilizatori. Cu comanda ls se pot vizualiza acești biți de protecție pentru fiecare fișier.

De exemplu, un fișier cu configurația **rwxrw-r--** poate fi citit de toată lumea, scris doar de proprietar și grup și executat doar de proprietar. Grupul de biți de protecție asociați unui fișier este completat cu încă doi:

— bitul 10 (set user **ID**, stabilire utilizator la execuție) cu semnificația: atunci cînd programul este executat de un utilizator oarecare, sistemul transferă identitatea proprietarului fișierului (programului) asupra acelui utilizator pe durata execuției. Această modificare temporară este destinată să asigure — în mod controlat — accesul la fișierele protejate. Mecanismul este utilizat de sistemul de operare însuși pentru utilitare, deținute de superuser, care, avînd setat acest drept, pot accesa în timpul execuției fișiere protejate inaccesibile în mod direct utilizatorilor neprivilegiați.

— bitul 11 (set group **ID**, stablirea grupului la execuție) cu o semnificație similară, transferul de drepturi făcîndu-se acum la nivel de grup.

Poziționarea drepturilor de acces ale unui fișier se face implicit la crearea acestuia și poate fi modificată cu comanda **chmod**.

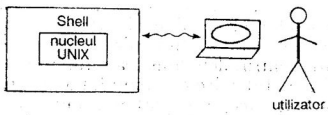


Fig. 2.4. Rolul interpretorului Shell.

2.5. INTERFAȚA UTILIZATOR

Sistemul UNIX constă din mai multe componente software, dintre care două au o poziție privilegiată: nucleul și Shell.

Nucleul reprezintă supervizorul sistemului, cel care planifică procesele și le execută în ordinea și la timpul cerut. Deși este o parte vitală a sistemului, nucleul este relativ mic comparativ cu dimensiunea totală a sistemului.

Cea mai mare parte din totalul codului executabil o reprezintă numeroasele utilitare (programe), incluzînd aici numeroasele compilatoare, editoare, procesoare de formatare a textelor etc.

Utilizatorul trebuie să hotărască și să comunice nucleului ce programe să lanseze în execuție, cum să le încarce, cum să apară ieșirea acestora. Programul Shell reprezintă unicul canal de comunicație utilizat în acest scop (fig. 2.4).

El este un interpretor de comenzi pentru UNIX, numit și Borne Shell.

2.5.1. Sintaxa comenzilor Shell

Inițializarea sesiunii de lucru a unui utilizator constă în introducerea de către acesta a numelui său și al parolei. După certificarea acestora cu informațiile pe care le are sistemul în fișierul **/etc/passwd**, se lansează automat interpretorul Shell, a cărui prezență este indicată de prompter-ul apărut pe ecran: **$**, pentru utilizatori neprivilegiați și *, pentru utilizatorul privilegiat (numit și superuser sau administrator).

Prompter-ul indică faptul să sistemul este gata să primească comenzi.

Din punctul de vedere al utilizatorului, Shell oferă o interfață complexă cu sistemul, pe două nivele:

(*1*) prelucrarea și execuția comenzilor: marea majoritate a acestora sînt programe independente de interpretor, prezente ca fișiere executabile în directoarele /bin, /usr/bin sau /etc.;

(*2*) dezvoltarea unor programe structurate, într-un limbaj numit Borne Shell, destinate unor aplicații în care sînt necesare înlănțuiri de programe după diverși algoritmi de execuție. Asupra programării în Shell vom reveni într-un alt paragraf.

Toate comenzile au următoarea sintaxă:

$ comanda {**argument***1*} {**argument***2*} ... {**argumentn**}

Comanda reprezintă numele programului ce urmează a fi executat, căutat în următoarea secvență:

(*a*) în directorul curent; dacă nu este găsit;

(*b*) în directorul /bin; dacă nu este găsit;

(*c*) în directorul /usr/bin; dacă nu este găsit;

(*d*) mesaj de eroare afișat de Shell.

Celelalte cîmpuri definesc parametrii comenzii și se numesc argumente; ele sînt separate prin blancuri. Dată fiind absența unui analizor pentru linia de comandă, argumentele vor fi transmise de interpretor programului asociat prin mecanismul specific limbajului C în care acesta este scris.

De exemplu, comanda **$ls-***1* va lista conținutul directorului curent. Argumentul -1 indică programul **ls** (prezent în /bin) că se dorește o listare cu informații complete. Pot fi introduse mai multe comenzi pe aceeași linie, separate prin „;".

De exemplu, comanda **$pwd;ls-***1* va afișa mai întîi numele directorului curent (**pwd**) și apoi va lista conținutul (**ls**).

În UNIX Shell este un program obișnuit (/bin/sh), tratat de sistem ca un proces neprivilegiat și independent de nucleu.

Interfațarea lui cu nucleul se face pe calea funcțiilor sistem.

După inițierea unei sesiuni de lucru la terminal, sistemul lansează un proces Shell pentru terminalul respectiv,

care alternează cererile de comenzi cu execuţia acestora, după următorul ciclu:

```
repetă
    scrie prompter „$" sau „#"
    citeşte linia de comandă
        *decodifică comanda
        *caută programul apelat
        *generează un proces „fiu" (directiva fork)
        *lansează în execuţie comanda (directiva exec)
        *aşteaptă terminare comandă (directiva wait)
până *comandă „bye" sau CTRL-D
```

2.5.2. Procese paralele

Uneori lansăm programe care durează mult şi care nu utilizează consola, în sensul că nu aşteaptă date de intrare de la terminal.

Pentru a folosi aceşti timpi morţi, Shell are o facilitate prin care se pot lansa mai multe procese în paralel (de obicei maxim 20), de către acelaşi utilizator.

În mod normal Shell execută comenzile secvenţial. Dacă însă o comandă se termină cu caracterul „&", atunci Shell va lansa programul în paralel şi va afişa imediat prompter-ul pentru introducerea unei noi comenzi. Acest mod de lucru în regim de multiproces, este indicat pentru acele programe care nu sînt interactive.

Aceste procese paralelele vor primi din partea sistemului un număr de identificare, care este folosit la o eventuală anulare a execuţiei, cu comanda **kill**.

Dacă vrem, de exemplu, să lansăm în paralel un program cu numele **prog** vom proceda astfel:

$ prog &.

Cu ajutorul comenzii **ps**, putem vedea apoi dacă procesul mai este în execuţie şi ce identificator (**pid**) i-a fost afectat:

```
$ps
  PID      TTY      TIME     CMD
  136      53       0 : 30   -sh
15388      53       0 : 14   prog
53350      53       0 : 05   ps
```


Cele 2 comenzi puteau fi date şi în aceeaşi linie:

$prog &**ps**

Deoarece sistemul nu anunţă terminarea unui proces paralel, se preferă lansarea acestuia sub forma unei secvenţe de tipul:

$ (comanda ;**echo** mesaj) &

care va afişa mesajul după terminarea programului indicat de comandă.

Dacă procesele paralele au totuşi de făcut intrări/ieşiri la terminal, pentru a nu perturba procesul care se desfăşoară conversaţional în acel moment, este indicat să se facă redirectarea fişierelor standard, aşa cum se va vedea în paragraful următor.

2.5.3. Fişierele standard de intrare şi ieşire

După iniţializarea sesiunii de lucru, Shell deschide 3 fişiere standard (0, 1, 2) pentru intrare, ieşire şi afişarea erorii.

Ele sînt atribuite terminalului (tastaturii şi ecranului) de la care se lucrează. În momentul în care Shell lansează în execuţie programul asociat unei anumite comenzi, acesta moşteneşte (ca proces „fiu") fişierele deschise, deci pe cele standard (fig. 2.5).

În acest fel programele devin independente faţă de dispozitivele fizice asociate de Shell fişierelor standard.

Interpretorul Shell însă permite redirectarea fişierelor standard de intrare/ieşire spre alte fişiere sau periferice.

a) Cu ajutorul „<" se redirectează *fişierul standard de intrare.*

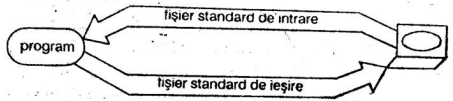


Fig. 2.5. Conectarea standard a fişierelor de intrare şi ieşire.

Acest lucru înseamnă că programul lansat de Shell nu va lua datele de intrare de la terminal, ci dintr-un fișier indicat după operatorul „<".

De exemplu, comanda:

$prog < fis

va redirecta fișierul de intrare al programului „prog" la fișierul „fis" pe disc (fig. 2.6).

Prin redirectarea intrării se poate lansa și un fișier de comenzi Shell:

$sh < fis-comenzi

unde „fis-comenzi" poate conține de exemplu: pwd; ls-1; ps.

b) Cu ajutorul „>" se redirectează fișierul standard de ieșire, ceea ce înseamnă că informația care trebuie afișată pe terminal va fi depusă în fișierul indicat după „>" (fig. 2.7.).

De exemplu, comanda

$ls-1,

va lista directorul curent la terminal, pe cînd

$ls-1 > fis,

va crea fișierul *fis* unde va depune informațiile directorul curent. Dacă fișierul există și vrem să se scrie în continuare,

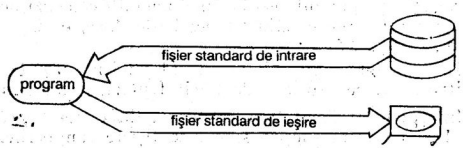


Fig. 2.6. Redirectarea intrării.

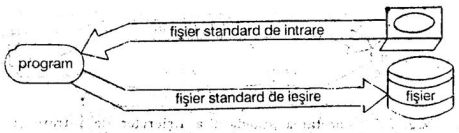


Fig. 2.7. Redirectarea ieșirii.

prin adăugare de informația deja existentă, putem folosi operatorul „>>".

$ls-1>>fis.

c) Un program poate fi lansat prin Shell cu redirectarea simultană a intrării și ieșirii, ca de exemplu:

$crypt<fis1>fis2,

comandă care execută cifrarea fișierului *fis1* depunînd rezultatul în fis2.

2.5.4. Conectarea programelor prin pipe

În concepția sistemului de operare UNIX, programele trebuie să fie de dimensiuni relativ mici, cu sarcini bine delimitate și care, plecînd de la anumite informații obținute pe fișierul standard de intrare vor genera rezultate pe fișierul standard de ieșire. Această structură permite însă ulterior conectarea acestor programe în cascadă pentru realizarea unor aplicații complexe; acest tip de conectare presupune legarea ieșirii unui program la intrarea următorului. Shell furnizează această facilitate pe care o numim **pipe**, indicată prin operatorul „|" și care redirectează ieșirea unui proces la intrarea altuia (fig. 2.8).

Există două metode de realizare a comunicării între procese:

a) **prin tehnica de redirectare,** cînd se folosește un fișier în care primul proces depune date pe care le va citi și prelucra cel de-al doilea proces;

b) **prin tehnica pipe,** cînd conectarea se face direct între cele două procese, fără a mai fi necesară existența unui fișier.

Să considerăm următorul exemplu: vrem să determinăm cîte intrări (fișiere) conține directorul /bin.

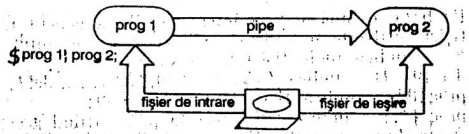


Fig. 2.8. Conectarea prin *pipe* a proceselor.

Pentru aceasta folosim comenzile **ls**, care listează directorul, și **wc**, care numără liniile unui fișier text:

```
$ls-1/bin>fișier
$xe-1<fișier
$rm fișier
```

Această secvență folosește tehnica de redirectare, fiind necesară la sfîrșit operația de ștergere a fișierului de manevră (comanda **rm**).

Folosind tehnica **pipe**, se poate obține același efect, mai simplu însă:

```
$ls-1/bin:wc-1.
```

Interpretorul Shell admite și conectări multiple a mai multor procese prin **pipe**.

Pentru implementarea conectării, interpretorul Shell, care este un program scris în limbaj C, folosește mecanismul de comunicare între procese oferit de nucleu prin intermediul directivei „pipe.

2.5.5. Numele fișierelor

Multe din argumentele liniilor de comandă sînt nume de fișiere.

Sintaxa acestora a fost deja precizată în unul din paragrafele anterioare. Shell permite însă și precizarea unor nume de fișiere, precum și a unor liste de nume de fișiere, folosind cîteva metacaractere:

* — orice șir de caractere (inclusiv șirul vid);
? — orice caracter;
[...] — o mulțime de caractere;
— — o secvență lexicografică.

Pe baza acestor metacaractere, Shell va genera nume de fișier, utilizate în cadrul comenzilor. De exemplu:

*.c — indică toate fișierele cu sufix *c*;
def [0—9] — indică fișierele def0, def1, ..., def9;
prog [09] — indică fișierele prog0, prog9;
prog [*?] — indică atît varianta prog* (fiind acceptat orice șir după „prog"), cît și prog? (fiind acceptat orice caracter după „prog").

Să considerăm un director cu următoarele fișiere:

cap1	cap2	cap3	cap4
a.c.	b.c.	cap3.c	cap4.c

Dacă folosim comanda:

```
$cat cap?
```

se vor afișa la terminal fișierele, cap2, cap3 și cap4.

Dacă însă tastăm comanda:

```
$cat cap*
```

la terminal vor apărea fișierele mai sus menționate, dar și cap3.c, cap4.c.

Utilizarea metacaracterelor facilitează introducerea rapidă a unor liste de nume de fișiere.

2.5.6. Comenzi Shell

Fiecare comandă în UNIX este un nume de program care va fi executat ca răspuns la introducerea ei. În cadrul acestui paragraf vom prezenta 60 dintre cele mai folosite comenzi ale Shell.

Ele vor fi grupate din punctul de vedere al funcționalității în mai multe categorii. Multe din aceste comenzi sînt de fapt niște utilitare, cu foarte multe facilități. Paragrafele următoare nu doresc să se substituie manualelor sistemului; ele își propun să facă doar o descriere sumară, cu rol de introducere în utilizarea sistemelor de tip UNIX. De aceea, din sintaxa completă a unor comenzi s-au ales doar acele opțiuni care sînt mai des folosite.

Cîmpurile opționale, conform uzanțelor, le-am intercalat între paranteze drepte [,]; unde trebuie să apară numele unor reprezentanți ai unei anumite entități (fișier, director etc.) se vor folosi parantezele ascuțite <, >, iar trei puncte ... vor indica că ceea ce este înaintea lor se poate repeta de un număr nedefinit de ori. De fiecare dată cînd va fi necesar se vor prezenta și exemple.

2.5.6.1. *Comenzi pentru gestiunea directoarelor*

1. pwd

funcția: tipărește numele (calea) directorului curent;
sintaxa: pwd

Această comandă afișează pe terminal calea completă (pornind din root) a directorului curent în care lucrează în acel moment utilizatorul. Ea este folosită în special pentru a vedea unde este plasat directorul curent în structura arborescentă a sistemului de fișiere.

2. **cd**

funcția: schimbă directorul curent;

sintaxa: **cd ⟨director⟩**

Se abandonează directorul curent și ⟨director⟩ devine noul director curent. De exemplu:

$cd/usr/eu — face ca directorul „eu" să devină director curent;

$cd../../noi — indică noul director curent pornind din vechiul director și urcînd în arbore;

$cd — dacă comanda nu are argument, directorul curent devine directorul „home", cel indicat în fișierul /etc/passwd ca director curent implicit.

3. **ls**

funcția: listează conținutul unor directoare;

sintaxa: **ls** [**opțiuni**] ⟨**fișier**⟩...

Se afișează la terminal conținutul oricărui fișier director din listă. Dacă fișierul nu este director, se listează anumite informații despre el, conform cu opțiunile. Dacă nu se precizează nici un nume de fișier, se listează conținutul directorului curent.

Opțiunile sînt precedate de caracterul minus (—) și pot fi combinate, mai multe la un loc. Cele mai importante sînt următoarele:

— **d** — se listează doar numele fișierelor de tip director;
— **i** — se indică i-numărul fiecărui fișier;
— **l** — listează mai multe informații despre fiecare fișier, nu numai numele;
— **g** — se indică identificatorul grupului proprietar al fișierului;
— **r** — se listează în ordine invers alfabetică (implicit listarea se face alfabetic);
— **s** — dimensiunea fișierelor se dă în număr de blocuri;
— **t** — se sortează fișierele înainte de afișare, după data ultimei modificări.

De exemplu:

$ls-1t — va lista conținutul directorului curent, sortat după data ultimei modificări;

$ls-1*.c — se vor lista informații despre toate fișierele din directorul curent care au sufixul .c;

$ls-ld /etc— se vor lista informații doar despre directoarele din /etc;

$ls — va lista toate fișierele din directorul curent;

$ls* — va lista toate fișierele din directorul curent, inclusiv numele subdirectoarelor și ale fișierelor conținute în ele.

Opțiunea -1 permite obținerea unor informații complete despre fișiere:

— cîmpul mod: — pentru fișier ordinar;
d pentru fișier director;
c pentru fișier special de tip caracter;
b pentru fișier special de tip bloc.

— privilegiul proprietarului: read (r), write (w), execute (x);
— privilegiul grupului: r, w, x;
— privilegiul celorlalți r, w, x;
— numărul de legături ale fișierului;
— numele proprietarului;
— numele grupului;
— dimensiunea fișierului (în octeți);
— data ultimei modificări;
— numele fișierului.

De exemplu, comanda:

$ls-lg/etc/rc va afișa:

```
-rw-rw-r--  1  root  adm  3060  ian  15 20:10/etc/rc
|   |   |   |   |     |     |     |            |
|   |   |   |   |     |     |    data         nume
|   |   |   |   |   grup dimensiune
|   |   |   |  proprietar
|   |   |  nr. de legături
|   | drepturile celorlalți
|  drepturile grupului
drepturile proprietarului
fișier ordinar
```

4. **mkdir**

funcția: creează un nou director
sintaxa: **mkdir ⟨director⟩**

Utilizatorul poate crea un nou director pe care-l leagă ca subdirector al directorului curent. Sistemul va seta pentru directorul creat toate drepturile (r, w, x) la toate categoriile de utilizatori.

5. **rmdir**

funcția: șterge un director
sintaxa: **rmdir ⟨director⟩**

Directorul indicat nu va fi șters decît dacă este gol (conține doar intrările „." și „.."). Va fi deci necesară o ștergere în prealabil a fișierelor componente cu comanda **rm**.

2.5.6.2 *Comenzi pentru fișiere text*

1. **cat**

funcția: concatenează apoi afișează fișiere text
sintaxa: **cat ⟨fișier ⟩**

Este una din comenzile Shell cele mai folosite. Ea permite ca mai multe fișiere să fie concatenate într-un singur șir de ieșire, care va fi listat la terminal . De exemplu, comanda

```
$cat fis1 fis2  >nou
```

va concatena cele două fișiere *fis1* și *fis2*, iar rezultatul se va depune în fișierul nou. Comanda este echivalentă cu:

```
$cat fis[12]  >nou
```

Dacă tastăm o comandă de forma:

```
$cat fis1
```

se va realiza doar afișarea la terminal a lui *fis1*.

O utilizare interesantă este și următoarea:

```
$cat /dev/mult >fis
```

care va permite crearea fișierului *fis* fără nici o informație în el (se copiază în el fișierul special al perifericului **nul**.

Cu ajutorul comenzii **cat** se poate crea rapid un fișier text, direct de la terminal, fără a mai fi necesară lansarea unui editor:

```
$ cat >fis.
```

În continuare, toate liniile se vor introduce în „fis", pînă la ^Z.

2. **pr**

funcția: tipărește fișiere text
sintaxa: **pr [opțiuni] ⟨fișier ⟩**

Această comandă este folosită pentru paginare, punerea titlului și listarea unuia sau mai multor fișiere text. Pagina începe cu un antet în care sînt puse data și numele fișierului. Cele mai importante opțiuni care se vor folosi sînt următoarele:

— **l ⟨n⟩** — setează lungimea paginii la *n* aaaa (implicit *n* = 66);
— **m** — listează toate fișierele simultan, cîte unul pe coloană;
— **⟨n⟩** — produce o ieșire formatată în **n** coloane;
— **⟨n⟩** — începe tipărire cu pagina **n**;
— **w⟨n⟩** — setează lățimea paginii la **n** caractere (implicit 72);

De exemplu:

```
$pr text1  text2  >text.pr
```

va pagina cele două fișiere *text1* și *text2*, iar rezultatul îl va dune în fișierul text. pr.

3. **lpr**

funcția: tipărește la imprimată fișiere text
sintaxa: **lpr [opțiuni] [⟨fișier ⟩...]**

Fișierele dimensionate sînt puse într-un fir de așteptare la imprimantă, unde vor fi listate pe rînd. Dacă vrem ca fișierul să fie paginat va trebui ca în prealabil el să fie prelucrat cu comanda pr. Dacă în configurație exi.tă mai multe imprimante, există mai multe versiuni ale utilizatorului lpr: mpr, vpr, ppr etc. Principalele opțiuni disponibile sînt:

— **c** — copiază fișierul imediat, pentru a-l proteja împotriva unor schimbări care pot apărea înainte de tipărire;
— **m** — emite o scrisoare (mesaj) cînd tipărirea s-a terminat;
— **r** — șterge fișierul după ce a fost plasat în firul de așteptare.

De exemplu, secvența de comenzi va tipări la imprimantă conținutul mai multor direetoare:

```
$ls-l /bin/usr/bin >fișier
$pr fișier >fișier.pr
```

```
$lpr fișier.pr
$rm fișier fișier.pr
```

4. **wc**

funcția: numără caractere, cuvinte și linii în fișiere text

sintaxa: **wc** [**opțiuni**] [⟨**fișier** ⟩...]

Este un utilitar care permite numărarea unor unități ale fișierelor text; cuvinte, caractere sau linii. Opțiunile pot fi:

— **w** — pentru numărarea cuvintelor;
— **c** — pentru numărarea caracterelor;
— **l** — pentru numărarea liniilor.

5. **diff**

funcția: găsește diferențele între două fișiere text

sintaxa: **diff** [**opțiuni**] ⟨**fișier 1** ⟩ ⟨**fișier 2** ⟩

Cele două fișiere sînt comparate linie cu linie. Diferențele dintre ele sînt afișate la terminal, indicîndu-se ce trebuie făcut astfel încît cele două fișiere să fie identice. Liniile precedate de ⟩= apar în fișierul al doilea și nu în primul, iar cele precedate de ⟨apar în primul fișier și nu în al doilea.

Se acceptă următoarele opțiuni:

— **b** — ignoră diferențele dintre linii determinate de blancuri;

— **e** — produce un șir de comenzi pentru editorul de prin care se recrează fișier—2 din fișier 1.

De exemplu, vrem să vedem dacă directorul curent a fost modificat de la ultima sa salvare în fisdir cu „ls ⟩fisdir":

```
ls diff fisdir
```

6. **sort**

funcția: sortează liniile în unul sau mai multe fișiere

sintaxa: **sort** [**opțiuni**] [**poz1**] [**poz2**] ... [**—o fișier—1**] ⟨**fișier** ⟩...

Liniile fișierului sînt sortate și scrise pe terminal sau în fișierul „fișier—1" (cu opțiunea —o). Implicit, sortarea este alfabetică. Se poate preciza poziția cheii de sortare în linie, prin indicele coloanei: poz1, poz1+1, ..., poz2—1. Numerotarea coloanelor începe de la 0. Principalele opțiuni sînt:

— **b** — ignorarea blancurilor la început de linie;
— **n** — sortare aritmetică;
— **r** — sortare descrescătoare;
— **c** — verifică dacă fișierul este sortat; dacă da, nu se afișează nimic la ieșire;
— **u** — suprimă liniile duplicate.

Exemple:

```
$sort —n +o —2 —o fis1 fis2
```

face sortarea fișierului fis2, după o cheie numerică situată în primele două coloane ale liniilor. Rezultatul sortării este depus în fișierul fis1.

7. **grep**

funcția: filtrează liniile unui fișier text, căutînd șabloane

sintaxa: **grep** [**opțiuni**] [**șir caractere**] ⟨**fișier** ⟩

Utilizatorul citește unul sau mai multe fișiere, căutînd un anumit șir de caractere în fiecare linie (șablon). Dacă nu se specifică opțiuni se vor afișa toate liniile care conțin șablonul.

Dintre principalele opțiuni amintim:

— **c** — produce un contor al liniilor în loc să afișeze liniile;

— **n** — afectează numere la liniile care nu conțin șablonul;

— **v** — caracterele mici din șablon vor fi identificate atît cu caractere mici, cît și cu caractere mari, în fișierul de intrare. De exemplu, comanda:

```
$grep —v editura capitol*
```

va afișa toate liniile care nu conțin cuvîntul „editura", din fișierele capitol... Comanda:

```
$who | grep tty30
```

va determina care utilizator este conectat la terminalul tty30. Prin comanda:

```
$ls—1 |grep '^d'
```

se conectează ieșirea lui ls la intrarea lui grep.

Din lista care cuprinde conținutul directorului curent se selectează cu grep doar liniile care încep cu caracterul 'd'; ca urmare se vor tipări doar informații despre directoare.

8. **tee**

funcția: duplică fișierul standard de intrare
sintaxa: **tee** [**opțiuni**] ⟨**fișier** ⟩ ...

Fișierul standard de intrare este transcris pe fișierul standard de ieșire, cu o extracopie depozitată în ⟨fișier ⟩. Există două opțiuni:

— a — adăugare la ⟨fișier ⟩ și nu scriere de la început;
— i — ignoră orice întrerupere în timpul transcrierii.

De exemplu în șirul de comenzi legate prin „pipe":

```
$ls—1 ¦tee fis1¦ pr¦ tee fis2¦ lpr
```

cele două comenzi **tee** au rolul de a salva, pe disc, în fișierele fis1 și fis2, fișierele intermediare care se transmit prin „pipe" de la un utilizator la altul.

9. **tail**

funcția: afișează sfîrșitul unui fișier text
sintaxa: **tail** [**dimensiune**] ⟨**fișier**⟩

Programul tail tipărește pe fișierul standard de ieșire ultima parte a lui ⟨fișier ⟩ (implicit ultimele 10 linii). Argumentul 'dimensiune' este folosit pentru a controla cît să se afișeze din ultima parte a fișierului:

— ⟨**n** ⟩ — se vor tipări ultimele ⟨**n** ⟩ linii ale fișierului
+ ⟨**n** ⟩ — se vor tipări toate liniile, mai puțin primele ⟨n ⟩.

De exemplu:

```
$tail—130 prog.c
```

va afișa ultimele 130 de linii ale fișierului 'prog.c'

Comanda:

```
$ls—1 |bin ¦tail—20
```

va afișa ultimele 20 de linii ale directorului '/bin'.

10. **comm**

funcția: găsește și afișează liniile comune în două fișiere text
sintaxa: **comm opțiuni** ⟨**fișier 1** ⟩ ⟨**fișier 2** ⟩

Cele două fișiere comparate vor trebui în prealabil or donate alfabetic (cu sort).

Ele vor fi citite, iar liniile vor fi afișate în trei variante:

1. cele care apar numai în ⟨fișier—1⟩ (opțiunea -1);
2. cele care apar numai în ⟨fișier—2⟩ (opțiunea -2);
3. cele care apar în ambele fișiere (opțiunea -3);

2.5.6.3. *Comenzi pentru gestiunea fișierelor*

1. **file**

funcția: determină tipul unui fișier
sintaxa: **file** ⟨**fișier** ⟩...

Comanda determină dacă fișierele indicate sînt de tip obiect, sursă sau oarecare. Pentru fișierele sursă încearcă să determine în ce limbaj de programare este scris.

2. **find**

funcția: găsește un fișier într-un subarbore
sintaxa: **find** ⟨**director** ⟩... ⟨**condiție** ⟩

Se caută în directoarele specificate și în toate subdirectoarele lor, pentru a se localiza toate fișierele care satisfac ⟨condiție ⟩. Numele fișierelor găsite este listat pe fișierul standard de ieșire. Condițiile se specifică prin opțiuni sau combinații logice de opțiuni. Ele se referă la numele fișierului, timpul de cînd a fost creat sau accesat etc. Cele mai importante opțiuni sînt:

— **name** ⟨**fișier** ⟩ — specifică numele fișierului căutat;
— **atime** ⟨**n** ⟩ — toate fișierele accesate în ultimele ⟨n ⟩ zile;
— **mtime** ⟨**n** ⟩ — toate fișierele modificate în ultimele ⟨**n** ⟩ zile;
— **print** — calea curentă este afișată;
— **exec** ⟨**comanda** ⟩ — specifică o comandă care va fi executată. Argumentul '{ }' este înlocuit cu calea curentă.

Sfîrșitul comenzii este marcat de șirul '/;';

— **group** ⟨**grup** ⟩ — specifică numele grupului căruia îi aparține fișierul.
— **type** ⟨**i** ⟩ — specifică tipul fișierului căutat:
  * f: ordinar;
  * d: director;
  * c: special caracter;
  * b: special bloc.

Exemple:

$find . -print

$find . -print

va tipări o listă cu toate fișierele din subarborele curent;

$find. -exec echo }/;

va avea același efect cu comanda anterioară;

$find / -nume *.obj -atime +7 -exec rm {}/;

va șterge toate fișierele de tip obj, care nu au mai fost accesate de mai mult de 7 zile.

Căutarea va porni din directorul rădăcină.

3. **od**

funcția: vidaj fișiere

sintaxa: **od** [**opțiuni**] ⟨**fișier** ⟩ [⟨**dimensiune** ⟩]

Acest utilitar este folosit pentru a produce vidajul unui anumit fișier. Acesta se poate face în octal (—0), în zecima (—d), în hexa (—x) sau în ASCII (—c). Argumentul ⟨dimensiune ⟩ indică în octeți sau în blocuri (cu sufix b) cît din fișier să fie vidat.

De exemplu, comanda:

$od —c fis1 +100

va face vidajul în ASCII a primilor 100 de octeți din fișierul *fis1*, iar

$od/dev/rp0 3b

va lista în octal primele trei blocuri din discul **rp0**.

4. **rm**

funcția: șterge fișiere

sintaxa: **rm** [**opțiuni**] ⟨**fișier** ⟩...

Șterge fișierele indicate din directorul curent. Principalele opțiuni permise sînt:

— **i** — întreabă utilizatorul înainte de ștergerea fiecărui fișier;

— **r** — întreabă utilizatorul fiind vorba de ștergerea unui subarbore;

— **f** — șterge fișierul fără a analiza dacă există dreptul de scriere la fișier.

De exemplu:

$rm —i*

va șterge toate fișierele din directorul curent, așteptînd confirmarea la fiecare din ele;

$rm —r dir

va șterge tot subarborele care pornește din directorul dir (fișiere și directoare).

5. **mv**

funcția: mută și redenumește un fișier

sintaxa: **mv** ⟨**fișier**—*1* ⟩ ⟨**fișier**—*2* ⟩ sau
mv ⟨**fișier** ⟩... ⟨**director** ⟩

În prima formă va muta ⟨fișier—1 ⟩ în ⟨fișier—2 ⟩ și va șterge ⟨fișier—i ⟩. Cele două fișiere trebuie să aibă același director „tată" (practic, se face redenumirea fișierului). În varianta a doua ⟨fișier ⟩ este mutat în ⟨director ⟩, fiind scos din directorul inițial.

6. **cp**

funcția: copiază un fișier

sintaxa: **cp** ⟨**fișier**—*1* ⟩ ⟨**fișier**—*2* ⟩ sau
cp ⟨**fișier** ⟩... ⟨**director** ⟩

În prima formă se face o copie a lui ⟨fișier—1 ⟩ care se va numi ⟨fișier—2 ⟩, fără ștergerea lui ⟨fișier—1 ⟩. În a doua formă se copiază mai multe fișiere într-un anumit director.

7. **ln**

funcția: creează o nouă intrare în director pentru un fișier

sintaxa: **ln** ⟨**fișier**—*1* ⟩ [⟨**fișier**—*2* ⟩]

Comanda se folosește pentru a da un anumit nume unui fișier existent. Dacă apar în comandă ambele argumente, atunci ⟨fișier—2 ⟩ este noul nume pentru ⟨fișier—1 ⟩.

Dacă ⟨fișier—2 ⟩ nu este menționat, atunci se creează în directorul curent un fișier cu numele identic cu ultima entitate a căii ⟨ fișier 1⟩ .

De exemplu:

$ln copie unix

va face ca fișierul 'copie' să aibă de acum înainte și un al doilea nume 'unix'; același fișier va putea fi referit prin oricare din aceste nume;

$ln ../etc/eu

va crea în directorul curent o intrare cu numele 'eu', care va desemna același fișier cu cel indicat prin calea din comandă.

8. **crypt**

funcția: cifrarea unui fișier

sintaxa: **crypt [⟨parola ⟩] [⟨fișier—1 ⟩] [⟨fișier—2 ⟩]**

Acest utilitar realizează protecția prin cifrare a unui fișier.

Mecanismul de cifrare se bazează pe simularea mașinii ENIGMA și va fi prezentat într-unul din paragrafele următoare.

Cheia de cifrare se obține din parola indicată. Dacă ea nu este furnizată în linia de comandă, programul o cere în mod explicit.

Acest utilitar asigură o protecție superioară mecanismului prezentat cu cei 111 biți de protecție. De exemplu,

$crypt abc ⟨fis—1 ⟩ ⟨fis—2 ⟩

va cifra fișierul fis 1 folosind o cheie derivată din parola /abc/; rezultatul cifrării se va depune în fișierul fis 2.

Descifrarea informațiilor din fișierul fis 2 se face astfel:

$crypt abc ⟨fis—2 ⟩ ⟨fis—1 ⟩

9. **cmp**

funcția: compară două fișiere

sintaxa: **cmp [opțiuni] ⟨fișier 1 ⟩ ⟨fișier 2 ⟩**

Se listează diferențele dintre cele două fișiere, afișîndu-se numărul liniei și al octetului unde acestea apar. Opțiunea afișează în zecimal numărul de ordine al octeților diferiți și octeții în sine.

10. **chown chgrp**

funcția: schimbă proprietarul sau grupul la care este afiliat un fișier

sintaxa: **chown ⟨nou proprietar ⟩ ⟨fișier ⟩...**
chgrp ⟨nou grup ⟩ ⟨fișier ⟩...

Aceste comenzi sînt folosite pentru a schimba proprietarul sau grupul aparținător pentru un fișier. Acest lucru permite transferul de fișiere între utilizatori sau schimbarea numelui utilizatorului (cel prezent în „/etc/passwd") sau al grupului (din „etc/group").

11. **chmod**

funcția: schimbă drepturile de acces ale unui fișier

sintaxa: **chmod ⟨drepturi ⟩ ⟨fișier ⟩...**

Comanda poate fi utilizată doar de proprietarul fișierului Precizarea drepturilor se face sub forma:

⟨cine ⟩ ⟨op ⟩ ⟨drept ⟩

Următoarea tabelă arată cum se construiesc drepturile:

cine		operator	
u	proprietar	—	retrage drept
g	grup	+	adaugă drept
o	ceilalți utilizatori	=	asigneză drept
a	toți		

Drepturile pot fi:

r — citire;
w — scriere;
x — execuție;
u — cele ale proprietarului;
g — cele ale grupului;
o — cele ale celorlalți utilizatori.

De exemplu:

$schmod a+r fis

alocă drept de citire tuturor utilizatorilor fișierului „fis";

$chmod go -rw fis

ridică privilegiile de citire și scriere la grup și la proprietar;

$chmod a=rwx fis

se asignează tuturor utilizatorilor drepturi r, w, x;

$chmod a—w .

va face imposibil pentru oricine să mai creeze fișiere în directorul curent.

12. **ar**

funcția: întreține o bibliotecă (arhivă)

sintaxa: **at ⟨cheie ⟩ [⟨reper ⟩] ⟨bibliotecă ⟩ ⟨fișier ⟩...**

Este un utilitar pentru inserarea, ștergerea sau înlocuirea de fișiere în biblioteci. Este utilizat în special pentru

crearea și ținerea la zi a bibliotecilor utilizate de încărcător. Numele bibliotecii este ⟨bibliotecă⟩, iar ⟨fișier⟩... este lista numelor de fișiere din bibliotecă asupra cărora se execută acțiunea; ⟨reper⟩ este numele unui fișier din bibliotecă relativ la poziția căruia se execută acțiunea. În fine, acțiunea se indică prin ⟨cheie⟩, care poate fi formată din una sau două caractere; caracterul inițial poate fi:

d — șterge fișierele din bibliotecă;
m — mută fișierele la sfîrșitul bibliotecii;
p — tipărește conținutul fișierelor indicate din bibliotecă;
q — adaugă fișierele la sfîrșitul bibliotecii și crează biblioteca dacă nu există;
r — înlocuiește fișierele indicate cu altele noi;
t — tipărește conținutul bibliotecii.

Cel de-al doilea caracter al cheii poate fi:

a — scrie ⟨fișier⟩ în arhivă după ⟨reper⟩;
b — scrie ⟨fișier⟩ în arhivă înainte de ⟨reper⟩;
v — dă informații complete despre acțiunile întreprinse.

De exemplu:

```
$ar q biblio p1.c, p2.c
```

va adăuga în /biblio/ cărțile /p1.c/ și /p2.c/ ; dacă biblioteca nu există, se va crea;

```
$ar tv biblio
```

tipărește conținutul bibliotecii cu informații complete despre fiecare fișier.

13. **tar**

funcția: întreține o bibliotecă (arhivă) pe bandă magnetică
sintaxa: **tar ⟨cheie⟩ [⟨fișier⟩...]**

Utilitarul este destinat să restaureze fișiere pe bandă magnetică. Acțiunile sînt controlate de ⟨cheie⟩, un șir de caractere, dintre care cel puțin o literă funcție și unul sau mai mulți modificatori. În continuare, se indică ce fișiere sau directoare se salvează sau se restaurează. Caracterele din cheie ce indică funcția cerută pot fi:

r — ⟨fișier⟩ va fi salvat la sfîrșitul benzii;
x — ⟨fișier⟩ va fi extras (restaurat) de pe bandă;

dacă nu se specifică ⟨fișier⟩ se restaurează tot conținutul benzii;
t — listează întregul conținut al arhivei;
c — crează o nouă arhivă pe bandă (implică și r);
u — ⟨fișier⟩ este adăugat pe bandă dacă a fost modificat de la ultima salvare.

Următoarele caractere se pot adăuga la litera funcției:
v — afișează fiecare fișier salvat/restaurat;
w — la fiecare fișier salvat/restaurat se cere confirmarea cu /y/ , din partea utilizatorului;
0, 1, ... 7 — se indică perifericul pe care este montată banda mt0, mt1, ..., mt7 (implicit mt1).

De exemplu:

```
$tar c0 fis1
```

crează o arhivă pe mt0, unde pune /fis1/ ;

```
$tar t0
```

listează conținutul arhivei;

```
$tar x0/usr/man/man3
```

scoate de pe bandă acest fișier și îl depune pe disc;

```
$tar ro fis1 fis2
```

adaugă pe bandă două fișiere;

```
$tar xvwo etc.
```

extrage de pe bandă cu afișare și cu cerere de confirmare.

2.5.6.4. *Comenzi pentru gestiunea volumelor*

1. **df**

funcția: afișează spațiul liber pe disc
sintaxa: **df [⟨fișier sistem⟩]**

Indică spațiul liber în blocuri de 512 octeți, care există pe un anumit volum unde se află montat un sistem de fișiere.

Dacă acesta nu este indicat, se afișează spațiul liber la toate sistemele de fișiere montate. De exemplu:

```
$df /dev/rp*
```

va lista spațiul liber existent pe toate volumele din configurație.

2. **dn**

funcția: tipărește sumarul privind utilizarea discului într-un subarbore
sintaxa: **dn [opțiune] ⟨director⟩**

Se determină modul de folosire a discului pentru toate fișierele și subdirectoarele din directorul indicat. Opțiunea -a va permite ce analiză să se facă asupra tuturor fișierelor, nu numai a subdirectoarelor, iar opțiunea -s va da doar totalul utilizării discului.

3. **mkfs**

funcția: creează (inițializează) o structură din sistemul de fișiere

sintaxa: **/etc/mkfs** ⟨**fișier special** ⟩ ⟨**prototip** ⟩

Comanda este folosită pentru formarea unui disc nou sau a unuia folosit, al cărui conținut se pierde; localizarea discului se face prin intermediul fișierului special asociat periferic ului.

Informația ⟨prototip ⟩ poate fi număr care să indice cîte blocuri de 512 octeți va conține volumul sau un fișier cu un șir de informații necesare formatării. Puterea de distrugere a lui mkfs este accesibilă doar administratorului, fiind plasată în fișierul /etc. Exemplu:

$/etc/mkfs/dev/rk1 4800.

4. **mount (umount)**

funcția: montarea /demontarea unui sistem de fișiere

sintaxa: **/etc/mount** ⟨**fișier special** ⟩ ⟨**director** ⟩ [—**r**] **/etc/umount** ⟨**director** ⟩

Arborele general al sistemului de fișiere este unic, avînd o singură rădăcină, directorul **root**.

Este însă posibil ca anumite grupe de fișiere să se găsească pe suporturi diferite care să fie introduse în sistem prin operația de montare. Fiecare volum conține un sistem propriu de fișiere, al cărui director rădăcină se poate atașa la orice director din sistemul general de fișiere. Opțiunea —r indică faptul că volumul este montat doar pentru acces de citire.

2.5.6.5 *Comenzi pentru gestiunea proceselor*

1. **ps**

funcția: afișează starea proceselor

sintaxa: **ps** [**opțiuni**] [⟨**proces** ⟩...]

Comanda dă informații despre procesele active în acel moment. Opțiunile cele mai folosite sînt:

a — toate procesele asociate cu terminalul;

l — listare informații complete;

n — toate procesele care nu sînt asociate cu terminalul.

Din informațiile afișate cu opțiunea l amintim:

— starea procesului (**s**): r — în execuție;
w — în așteptare;
s — suspendat
t — oprit;
z — terminat;
o — inexistent;

— identificatorul utilizatorului (*uid*);

— identificatorul procesului (*ppid*);

— identificatorul „tatălui" procesului (*ppid*);

— terminalul care controlează procesul (*TTY*);

— timpul de execuție cumulat (*TIME*).

2. **nice**

funcția: încarcă un task cu prioritate redusă

sintaxa: **nice** [— ⟨**prioritate** ⟩] **comanda** [**argumente**]

Este folosită pentru lansarea unui proces (*task*) în paralel, cu prioritate scăzută. Prioritatea este un număr, numerele mici înseamnă **prioritate mare**.

Numai administratorul poate executa procese cu prioritate ridicată (—10), negativul însemnînd o prioritate ridicată.

Prioritățile date prin **nice** trebuie să fie pînă la 20.

Dacă prioritatea nu este precizată, se incrementează cu 10 în mod implicit prioritatea afectată de sistem. **Exemple**:

$nice — 19 prog1 &

lansează un program în paralel cu minimum de prioritate;

$nice — 19 prog1,&

lansează același program cu maximum de prioritate (negativă).

3. **kill**

funcția: oprire proces în curs de execuție

sintaxa: **kill** ⟨**proces** ⟩

Un proces în curs de execuție poate fi oprit folosind identificatorul său (*pid*) ca argument în comandă. De obicei, idenfificatorul este obținut prin comanda **ps**.

4. **sleep**

funcția: suspendă temporar execuția unui proces

sintaxa: **sleep** ⟨**timp** ⟩

Timpul este exprimat în secunde.

5. **time**

funcția: furnizează timpul de execuție a unei comenzi

sintaxa: **time** ⟨**comanda** ⟩

Se lansează în execuție ⟨comandă ⟩, iar la sfîrșitul execuției se afișează timpul de execuție și timpul UC în secunde

6. **sh**

funcția: lansează în execuție interpretorul Shell

sintaxa: **sh** ⟨**fișier** ⟩

Permite lansarea în execuție a unui fișier de comenzi Shell.

8. **at**

funcția: lansează în execuție un fișier de comenzi Shell la un timp specificat

sintaxa: **at** ⟨**timp** ⟩ [⟨**zi** ⟩] ⟨**fișier** ⟩

Timpul se specifică în ore și minute. Ziua se specifică sub forma datei. De exemplu:

$at 17 jan2 prog. sh

va lansa la ora 17, în ziua de 2 ianuarie programul Shell „prog. sh".

9. **test**

funcția: comandă condițională

sintaxa: **test** ⟨**expresie** ⟩

Se testează ⟨expresie ⟩, iar dacă valoarea este true se va returna 0 la ieșire. Condițiile se referă la fișiere, directoare, șiruri etc.

Pentru construirea expresiei se folosesc următoarele primitive:

— **r** ⟨**fișier** ⟩ — testează dacă fișierul există și este accesibil la citire;

— **w** ⟨**fișier** ⟩ — testează dacă fișierul există și este accesibil la scriere;

— **f** ⟨**fișier** ⟩ — testează dacă fișierul există și nu este director;

— **d** ⟨**fișier** ⟩ — testează dacă fișierul există și este director;

— **s** ⟨**fișier** ⟩ — testează dacă fișierul există și are dimensiunea mai mare ca zero;

— **z** ⟨**șir** ⟩ — testează dacă lungimea șirului este 0;

— **n** ⟨**șir** ⟩ — testează dacă lungimea șirului este diferită de 0;

șir 1=**șir** 2 — testează dacă două șiruri sînt egale;

șir 1! =**șir** 2 — testează dacă două șiruri sînt diferite.

2.5.6.6. *Comenzi informaționale și pentru terminale*

1. **man**

funcția: afișează capitole din Manualul Programatorului

sintaxa: **man** [**opțiuni**] [⟨**capitol** ⟩] ⟨ **secțiune** ⟩

Permite afișarea manualului sistemului UNIX pentru consultarea lor în timpul sesiunii de lucru; ⟨capitol ⟩ este un număr de capitol în care se caută ⟨secțiune ⟩ (în caz contrar căutarea se face în tot manualul). Principala opțiune admisă este —w, care cere afișarea doar a căii pînă la secțiunea indicată, fără text. De exemplu:

$man man

va afișa secțiunea din manual corespunzătoare man.

2. **cal**

funcția: permite afișarea calendarului

sintaxa: **cal** [**luna**] **an**

3 **date**

funcția: afișează data și ora

sintaxa: **date**

4. **who**

funcția: afișează numele utilizatorilor conectați la sistem

sintaxa: who [am I]

În varianta cu argument se listează datele utilizatorului însuși (numele terminalului, numele utilizatorului etc.).

5. **echo**

funcția: afișează argumente

sintaxa: **echo** [—n] **argumente**

Se folosește pentru afișarea argumentelor folosite de Shell, în cazul cînd șirurile ce denotă argumente devin complicate.

Opțiunea —n suprimă caracterul **newline** adăugat în mod normal lui echo.

6. **passwd**

funcția: schimbă parola sau o instalează

sintaxa: **passwd**

Utilizatorul este întrebat atît despre vechea parolă, cît și despre cea nouă.

Ultima trebuie tastată de două ori pentru a preveni erorile.

Parola trebuie să conțină cel puțin patru caractere. Asupra protecției prin parolă vom reveni.

7. **tty**

funcția: afișează numele terminalului

sintaxa: **tty**

8. **stty**

funcția: setează caracteristicile terminalului

sintaxa: **stty** [**opțiune**]

Se stabilesc caracteristicile terminalului pe care se lucrează:

— paritatea (opțiunea **even**, pentru paritate pară și **odd** pentru paritate impară);

— ecou (opțiunea **echo**);

— transformare litere mari în litere mici (opțiunea **lcase**;

— viteze de transmisie (300, 1200 etc);

— stabilirea caracterelor de control în buferele de linie (opțiunile **ek**, **erase c**, **kill c**);

9. **login**

funcția: intrare în sistem

sintaxa: **login** [⟨**utilizator** ⟩]

Comanda este folosită pentru intrarea în sistem a unui nou utilizator, fără ca vechiul utilizator să-și fi terminat sesiunea.

Se cere și se verifică parola utilizatorului.

10. **mail**

funcția: emite/recepționează poșta electronică

sintaxa: **mail** ⟨**utilizator** ⟩... **(la emisie poștă)**
mail [**opțiune**] **(la recepție poștă)**

În prima formă comanda **mail** emite poșta electronică altor utilizatori din sistem. Tot ce urmează după comandă, pînă la linia ce conține doar caracterul „.", este considerat ca mesaj ce va fi transmis. Mesajul va fi pus în fișierul „cutie poștală" a destinatarului, în directorul /usr/spool/ /mail.

În cazul unor mesaje lungi este indicată construirea lor cu editorul și emiterea lor prin redirectarea fișierului standard de intrare:

$mail ⟨utilizator ⟩ < ⟨fișier ⟩

A doua formă a comenzii permite „citirea" corespondenței sosite pentru un utilizator. Ultimul mesaj introdus în „cutia cu scrisori" va fi citit primul. Opțiunile permit:

— citire mesaj cu mesaj;

— ștergerea din fișier a unui mesaj (d);

— memorarea într-un fișier a celor mai importante (-s ⟨fișier ⟩);

— repetarea afișării unui mesaj (r), etc.

De exemplu, pentru a transmite o scrisoare utilizatorilor ȘERBAN și MONICA vom tasta:

$mail șerban monica
_____ } mesaj
_____

11. **write**

funcția: trimite imediat un mesaj la un utilizator

sintaxa: **write** ⟨**utilizator** ⟩ [⟨**terminal** ⟩]

Permite comunicația directă între utilizatori și nu pe principiul „cutiei poștale" ca în cazul **mail**. Toate liniile

ce vor fi tastate după **write** vor fi imediat transmise la recepție, pînă la tastarea lui CTRL-Z.

Pentru comunicații bilaterale se poate folosi un protocol în care fiecare element de dialog se termină cu o linie cu caracterul „c".Acest lucru îi indică partenerului că acum poate emite el mesajul, care va fi terminat din nou cu„o" ș.a.m.d., pînă cînd unul din parteneri termină cu CTRL-Z.

2.6. LIMBAJUL DE PROGRAMARE SHELL

Shell reprezintă poate cel mai important program al sistemului UNIX. El nu este numai un limbaj de comandă, ci și un adevărat-limbaj de programare de nivel înalt. El dispune de posibilități specifice limbajelor structurate, cum ar fi variabile, proceduri, structuri de control. De asemenea, Shell prezintă facilități de macroprogramare și de definire a tratării unor evenimente externe, prin instrucțiunea **trap.** Versiunea 7 a lui Shell se mai numește Bourne Shell, după creatorul său. S.R. Bourne.

Ea se poate caracteriza ca fiind în același timp un interpretor interactiv de comenzi și un interpretor de limbaj de comandă de nivel înalt.

2.6.1. Execuția programelor Shell

Vom numi program Shell sau fișier de comenzi orice comandă sau secvență de comenzi memorată într-un fișier disc.

În mod uzual vom folosi termenul de fișier de comenzi pentru acele fișiere care conțin simple secvențe de comenzi și termenul de program Shell pentru fișierele care conțin aranjamente de comenzi mai complicate și care folosesc de obicei comenzi condiționale pentru realizarea deciziilor și ciclurilor.

Apelul unui program Shell (fișier de comenzi) se poate face în trei moduri:

1. prin redirectarea intrării la Shell:
   $sh < ⟨fișier ⟩
   unde ⟨fișier ⟩ este numele fișierului disc care conține comenzile Shell;

2. prin precizarea directă a fișierului cu comenzi Shell:
   $sh ⟨fișier ⟩
3. prin lansarea în execuție a unui fișier text
   $ ⟨fișier ⟩G
   unde ⟨fișier ⟩ conține comenzile programului Shell. Acest fișier text trebuie creat în prealabil cu editorul și trebuie să i se dea privilegii de execuție cu comanda chmod:
   $chmod a+x ⟨fișier ⟩

Să observăm faptul că fiecare utilizator posedă de obicei în directorul „home" (asociat utilizatorului) un fișier de comenzi numit /.profile/, care se execută automat ori de cîte ori se inițializează o sesiune de lucru. Rolul său este de a inițializa anumite caracteristici ale terminalului și eventual de a afișa unele informații necesare la început de lucru (data, ora, poșta electronică sosită, spațiul liber pe disc, conținutul directorului („home"). De exemplu, putem crea un astfel de fișier care să conțină următoarele comenzi:

```
stty -lcase scape
date
mail
tty
df /dev/rp*
ls-l
echo salut și succes
```

2.6.2. Variabile Shell

În cadrul limbajului de programare variabilele sînt folosite pentru memorarea unor valori care se schimbă pe timpul execuției.

În cazul limbajului Shell variabilele sînt definite prin nume și pot conține ca valoare un șir de caractere.

Atribuirea unei valori unei variabile se face cu comanda de asignare:

⟨variabila ⟩ = ⟨valoare ⟩

De exemplu:

a = UNIX

va asigna variabilei cu numele *a* șirul „UNIX".

Dacă şirul asignat conţine şi spaţii, asignarea se face astfel:

b = „SISTEMUL UNIX"

Referirea unei variabile se face prin numele ei, precedat de caracterul $, caz în care loc substituţia variabilei cu valoarea sa. De exemplu:

echo $b

va duce la afişarea textului SISTEMUL UNIX.

Există cîteva categorii speciale de variabile, cum ar fi:

— variabile accesibile doar la citire, declarate cu cuvîntul read only;

— variabile valabile şi pentru procesele „fii" (globale), declarate cu cuvîntul export.

O categorie aparte de variabile sînt cele predefinite Iniţializate şi actualizate de interpretor, ele sînt de două categorii:

1. variabile read only actualizate de interpretor:
   $? — conţine codul de revenire al ultimei comenzi executate;
   $$ — conţine identificatorul de proces al lui Shell (pid);
   $! — conţine identificatorul ultimului proces lansat în paralel;
   $# — conţine numărul de argumente poziţionale pentru Shell;
   $0 — conţine numele comenzii executate de Shell;
   $1, $2, ... — conţin argumentele poziţionale la apelul unei comenzi Shell;
2. variabile iniţializate la intrarea în sesiune:
   $HOME — conţine numele directorului „home" afectat utilizatorului la intrare în sesiune;
   $PATH — conţin căile în care Shell caută programele (de regulă /usr/bin şi /bin);
   $PS1 — defineşte prompter-ul asociat lui Shell (implicit $);
   $PS2 — defineşte al doilea prompter (implicit >) folosit atunci cînd o comandă se continuă pe o altă linie;
   $TERM — conţine numele terminalului pe care se lucrează.

De exemplu, se poate face schimbarea prompter-ului prin asignarea:

$PS1 = „aştept comanda >"

cînd şirul dintre ghilimele va înlocui prompter-ul „$".

2.6.3. Proceduri Shell

Apelul unei proceduri este identic cu cel al unei comenzi:

$ ⟨procedura⟩ ⟨arg1⟩ ...

Procedura Shell corespunde unui fişier de comenzi. Parametrii transmişi procedurii sînt identificaţi cu variabilele predefinite $0 (pentru numele comenzii) şi $1, $2, ... pentru parametrii poziţionali. Variabilele $# conţin numărul acestor parametri.

Procedurile Shell se pot apela recursiv, aşa cum vom exemplifica în unul din paragrafele următoare. De exemplu,

$ls — l/usr

va apela comanda ls cu parametrii $0=„ls", $1=„—l" şi $2=„/usr".

Contorul $# conţine valoarea 3.

2.6.4. Instrucţiuni de control

Execuţia programelor Shell presupune analiza unor structuri mai complexe. în care se execută anumite secvenţe sau altele în funcţie de verificarea anumitor condiţii.

Noţiunea de instrucţiune din limbajul de programare este înlocuită aici cu cea de comandă. În urma execuţiei, fiecare comandă furnizează un cod de retur, care va fi 0 (adevărat) în cazul execuţiei corecte şi 1 (fals) în caz contrar.

Borne Shell dispune de doi operatori condiţionali cu ajutorul cărora se creează suite de comenzi, numite şi liste.

Există trei tipuri de comenzi în programele Shell:

1. **comenzi simple** — formate din comandă plus argumente;
2. **comenzi în „pipe"** — formate din grupe de comenzi simple conectate în cascadă prin operatorul „I";

3. **liste** — care sînt secvențe de comenzi simple sau comenzi în „pipe", separate prin:

— „;" sau **newline** — indicînd execuția secvențială;

— „&&" — indicînd execuția celei de a doua comenzi doar dacă prima returnează 1 (nu a mers bine);

— „||" — indicînd execuția celei de a doua comenzi doar dacă prima returnează o (a mers bine);

— „&" — indicînd execuția în paralel a comenzii.

Structura de bază a programelor Shell este lista. De exemplu, lista:

```
test-d /usr/eu && echo este director
```

va afișa mesajul „este director" dacă fișierul /usr/eu este de tip director;

```
test-d /usr/eu || nu este director
```

va afișa mesaj dacă /usr/eu nu este de tip director.

2.6.4.1 *Decizia*

Instrucțiunea permite implementarea deciziei și ca urmare ramificarea programului în funcție de anumite condiții.

Sintaxa este următoare:

```
if ⟨lista—1 ⟩              if ⟨lista—1 ⟩
  then ⟨lista—2 ⟩             then ⟨lista—2 ⟩
else ⟨lista—3 ⟩           elif ⟨lista—4 ⟩
fi                             then ⟨lista—5 ⟩
                               else ⟨lista—6 ⟩
                               fi
        forma 1                    forma 2
```

Cuvintele **if, else, then, elif, fi** cînt cuvinte cheie care marchează elementele structurii.

În forma 1 se execută ⟨lista—1 ⟩, iar dacă la sfîrșit codul este 0, se execută ⟨lista—2 ⟩, în care codul va executa ⟨lista—3 ⟩.

În forma 2 se pot testa mai multe condiții: dacă ⟨lista—1 ⟩ se termina cu 0, se va executa ⟨lista—2 ⟩, dacă ⟨lista—4 ⟩ se termină cu 0 se va executa ⟨lista—5 ⟩, în caz contrar se execută ⟨lista—6 ⟩.

Instrucțiunea **case** are sintaxa:

```
case ⟨sir ⟩ in
        ⟨șablon—1 ⟩ ⟨lista—1 ⟩
        ⟨șablon—2 ⟩ ⟨lista—2 ⟩
        ...
esac
```

Se implementează decizia multiplă. Se verifică ⟨sir ⟩ dacă este identic cu unul din ⟨șablon—i ⟩; în caz afirmativ se execută ⟨lista—i ⟩ atașată șablonului.

2.6.4.2. *Ciclul*

Ciclul reprezintă repetarea unei secvențe de liste de mai multe ori pînă la îndeplinirea unei condiții, testată la finalul ciclului.

Ciclul **while** are sintaxa:

```
while ⟨lista—1 ⟩
do    ⟨lista—2 ⟩
done
```

Se va executa ⟨lista—2 ⟩ în mod repetat, cît timp ⟨lista—1 ⟩ dă valoarea **adevărat** (0).

Se va ieși din ciclu cu valoarea false.

Ciclul **until** are sintaxa:

```
until ⟨lista—1 ⟩
do    ⟨lista—2 ⟩
done
```

Se va executa ⟨lista—2 ⟩ în mod repetat, cît timp ⟨lista—1 ⟩ da valoarea **fals** (1).

Se va ieși din ciclu cu valoarea adevărat.

Ciclul **for** are sintaxa:

```
for ⟨variabila ⟩ [în ⟨val—1 ⟩ ...]
    do ⟨lista ⟩
done
```

Se execută ⟨lista ⟩ în mod repetat ⟨variabila ⟩ luînd la fiecare ciclu, pe rînd, una din valorile ⟨val—1 ⟩, ...

În cazul în care lipsește **in**, ⟨variabila ⟩ ia succesiv valorile date ca parametri ai procedurii.

2.6.4.3 *Comenzi speciale*

În afara comenzilor Shell prezentate în § 2.5.6, interpretorul Shell mai recunoaşte şi o serie de comenzi speciale, dintre care amintim următoarele:

fişier — citeşte şi execută comenzi dintr-un ⟨fişier⟩ şi apoi revine în fişierul iniţial;

break — permite ieşirea din ciclu înainte de îndeplinirea condiţiei;

continue — permite reluarea ciclului cu următoarea iteraţie ,înainte de terminarea iteraţiei curente;

cd — schimbare director curent;

exec ⟨arg⟩ ... — comenzile specificate ca argumente sînt executate de Shell în loc să se creeze procese separate de execuţie;

shift — realizează deplasarea argumentelor cu o poziţie la stînga ($2— >$1, $3—> $2 etc.);

times — afişează timpul consumat de un proces;

wait [⟨pid⟩] — permite sincronizarea unui proces cu sfîrşitul procesului cu ⟨pid⟩-ul indicat. Dacă acest argument lipseşte se aşteaptă sfîrşitul tuturor proceselor „fii";

set [⟨arg⟩] — permite afişarea tuturor variabilelor (fără argument), afişarea comenzilor şi a argumentelor lor pe măsură ce sînt executate (argument —x) etc.;

expr ⟨expresie⟩ — permite evaluarea unei expresii.

În exemplele pe care le vom prezenta în paragraful 2.6.5, vom indica modul de utilizare a unora din aceste comenzi speciale.

2.6.4.4 *Substituţia comenzilor*

Atunci cînd într-un program Shell o comandă este încadrată de caracterul ' (accent grav), ea este executată de Shell, iar rezultatul (textul) va substitui locul comenzii în program.

De exemplu, comanda:

```
acum = 'date'
```

va atribui variabilei *acum* rezultatul execuţiei comenzii date, care este şirul de caractere ce conţine data curentă.

Apoi se poate afişa această informaţie cu:

```
echo $acum
```

Substituţia este folosită numai pentru a putea construi expresii aritmetice cu variabile Shell:

```
contor = 1
contor = 'expr $contor +1'
```

Se va executa mai întîi expr, care va calcula expresia, al cărei rezultat (2) va fi asignat lui contor.

În general, execuţia unui program Shell este precedată de cîteva faze premergătoare, dintre care amintim:

— substituţia comenzilor încadrate cu accente grave (') în textul rezultat;

— substituţia variabilelor cu valoarea lor;

— generarea numelor de fişiere pe baza metacaracterelor *, ?, [,].

2.6.5. Exemple de programe shell

În general, programatorii sînt dispuşi să lucreze în limbajele convenţionale.

Utilizarea limbajului Shell poate însă aduce de multe ori o importantă simplificare a efortului de programare, cel puţin în cîteva categorii de probleme:

— aplicaţii a căror rezolvare implică mai multe operaţii care sînt printre comenzile Shell;

— aplicaţii care manipulează date sub formă de linii text sau fişiere text;

— aplicaţii care implică consultarea directoarelor şi parcurgerea arborelui sistemului de fişiere.

Pentru a sublinia puterea şi eficienţa limbajului Shell vom prezenta mai multe exemple comentate:

EXEMPLUL 1

```
while test-r fişier
  do sleep 5
done
```

Programul testează dacă „fişier" este accesibil la citire; în caz afirmativ programul se suspendă 5 secunde.

EXEMPLUL 2

```
until test-r fișier
  do sleep 5
done
```

Programul se suspendă cîte 5 secunde ori de cîte ori „fișier" nu este accesibil la citire.

Ieșirea din ciclu se face pe condiție adevărată.

EXEMPLUL 3

```
date > / dev/lp
    if test-r erori
      then cat erori > / dev / lp
                rm erori
      else echo „nu sînt erori" > / dev / lp
    fi
```

Programul începe prin a scrie data la imprimantă (/dev/lp).

Apoi va testa dacă există și este accesibil la citire fișierul „erori". Dacă există, îl va lista la imprimantă și apoi îl va șterge. Dacă nu există, se va tipări mesajul „nu sînt erori".

EXEMPLUL 4

```
      contor = $ #
cmd = echo
while test $contor — gt 0
    do
              cmd = "$cmd / $$contor"
              cmd = 'expr $contor — 1'
done
 eval $cmd
```

Programul realizează tipărirea argumentelor cu care a fost apelat, în ordine inversă.

Se execută cîte un ciclu pentru fiecare argument, începînd cu ultimul.

Argumentul de prelucrat este indicat de variabila **contor**, care pleacă de la valoarea **$ #** (numărul de argumente)

și se decrementează la fiecare ciclu. Programul va construi o comandă **echo**, căreia îi va adăuga argumentele în ordine inversă, prin instrucțiunea: **cmd = „$cmd / $$contor"**; în cadrul ei șirul /$$ cmd are ca scop să creeze un text format din **$** și valoarea curentă a variabilei **$cmd** (/ semnifică faptul că **$** care urmează trebuie luat ca atare și nu ca un caracter de început al variabilei).

În final, comanda **cval** permite execuția argumentelor sale care sînt reprezentate de comanda **echo**.

EXEMPLUL 5

```
contor = $ #
cmd = echo
while true
    do
          cmd = „cmd / $$contor"
          contor = 'expr $ contor — 1'
          if test $contor — eq 0
              then break
          fi
done
eval $cmh
```

Programul face listarea argumentelor în ordine inversă, utilizînd comanda **break** pentru ieșire forțată din ciclu.

EXEMPLUL 6

```
contor = $ #
cmd = echo
while true
      do
          cmd = „$ cmd / $$contor"
          contor = 'expr $contor — 1'
          if test $contor-gt 0
            then continue
          fi
          eval $ cmd
          exit
    done
```

Programul listează de asemenea argumentele, exemplificînd utilizarea comenzii **continue.**

EXEMPLUL 7

```
list = „ „
for arg
      do $ list = „$arg $ list"
      done
echo $ list
```

Programul va face listarea argumentelor cu care a fost apelat, în ordine inversă. Se construiește șirul **list** care este apoi afișat cu comanda **echo.**

EXEMPLUL 8

```
for arg
      do
            grep $arg fis —vechi  › › fis—nou
      done
```

Programul își propune actualizarea unui fișier de conturi (fis — vechi) extrăgînd din el articole care conțin anumite conturi cu ajutorul utilitarului **grep.** Rezultatul va fi redirectat în noul fișier de conturi (fis — nou).

Conturile sosesc ca argumente ale programului la lansare. În ciclul **for**, arg va parcurge toate argumentele poziționale.

EXEMPLUL 9

```
if test $ # — eq 0
   then ls-l | grep | '^d'
   else
      for arg
            do
                  ls-l $arg | grep | '^d'
            done
fi
```

Programul va face listarea unor directoare; dacă lipsesc argumentele ($ # = 0) se face listarea din directorul curent doar a liniilor care conțin informații despre director (selectate cu **grep**). Dacă programul se lansează cu argumente, se vor lista subdirectoarele din directoarele indicate ca **argumente.**

EXEMPLUL 10

```
if test $ # -eq 0
      then ls-l | grep '^d'
      else for i
            do
            for j in $i/*
               do
               if test -d $j
                  then echo $j
               fi
            done
      done
fi
```

Programul realizează de asemenea listarea subdirectoarelor fie din directorul curent, fie din directoarele venite ca argumente la lansarea programului. Pentru fiecare director din argumente (i), pentru fiecare fișier din acesta (j), se testează dacă este de tip director (test-d) și în caz afirmativ se listează numele său (echo).

EXEMPLUL 11

```
if test $   -eq 0
   then nume =.
   else nume = $1
fi
cd $1
for i in *
   do
      if test -d $i
         then echo $nume / $i
              (cd $i; /prog1. sh $nume/$i)
      fi
   done
```

Programul își propune listarea tuturor directoarelor dintr-un subarbore, plecînd fie din directorul curent (cînd nu sînt argumente), fie din directorul indicat ca argument.

În prima parte a programului se inițializează variabila 'nume' fie cu numele directorului primit ca argument fie cu directorul curent (.).

Apoi se execută un ciclu pentru fiecare fișier din acel director; dacă este de tip director (test-d), atunci numele său este afișat (echo) și se apelează „(cd $i; /prog1. sh $nume/$i)".

Atunci cînd într-un program apare o listă între paranteze, comenzile vor fi executate în subshell.

Aici se va schimba directorul (cd) și apoi se va apela recursiv programul prog1. sh pe el însuși, cu alt argument (se presupune că programul Shell a fost depus pe disc în fișierul 'prog1. sh'). Cînd se termină procesul subshell, controlul revine procesului apelant.

Numărul de niveluri de apel recursiv nu este limitat.

2.7. INTERFAȚA PROGRAMATOR

Sistemul UNIX oferă numeroase facilități pentru dezvoltarea de programe superioare altor sisteme de operare. Limbajul pilon al sistemului este C, deoarece în el este scrisă cea mai mare parte a sistemului însuși. Este un limbaj structurat de nivel înalt, mult asemănător cu PASCAL, avînd însă și unele facilități de limbaj de asamblare. Este primul limbaj care îmbină în mod fericit avantajele limbajelor de nivel înalt cu puterea unor limbaje apropiate de arhitectura mașinii. Alte limbaje disponibile sub UNIX sînt: FORTRAN 77 (o variantă structurată de FORTRAN), BASIC, PASCAL (varianta semioficială realizată la Universitatea Berkeley), PROLOG, ADA și asamblorul corespunzător limbajului mașină.

De asemenea, UNIX dispune de editoare de texte care permit crearea unor fișiere text pe disc: editorul standard **ed** și editorul de tip ecran **vi** sînt cele mai folosite.

De o mare importanță pentru editarea unor lucrări sau manuale sînt și utilitarele pentru formatarea textelor; **Wroff, troff, tbl.** Vom prezenta în continuarea acestui paragraf cele mai importante facilități de programare disponibile sub UNIX.

2.7.1. Editorul de texte ed

Editorul de bază al sistemului UNIX este **ed** — un editor **orientat linie**, adică vede segmentele de text ca fiind compuse din linii individuale; **ed** este un program interactiv destinat creării, adăugării sau modificării în fișiere text. În mod normal, **ed** citește un fișier, îl modifică după dorința utilizatorului și apoi îl scrie într-un alt fișier (sau în cel original). Editorul manipulează informația din textul sursă făcînd o copie într-o zonă de memorie numită **buffer de editare**, care poate fi integrat în memoria internă (la fișiere de dimensiuni mai mici) sau în memoria internă și pe disc (la fișiere de dimensiuni mari). În fig. 2.9 este reprezentat modul de editare a programelor.

Bufferul de editare este văzut de **ed** că o mulțime de linii, separate prin caracterul newline. Caracterul de sfîrșit de fișier este end-of-file.

* **Lansarea** editorului se face cu comanda:

$ed ⟨fișier⟩,

unde ⟨fișier⟩ este numele fișierului editat.

* Operațiile executate de editor sînt indicate interactiv prin comenzi introduse de utilizator. **Sintaxa** unei linii de comandă este următoarea:

[⟨linie⟩], [⟨linie⟩] ⟨verb⟩

Informațiile ⟨linie⟩ au efect de poziționare în bufferul de editare. Comanda propriu-zisă este dată prin ⟨verb⟩ și poate fi:

a (append) — adaugă linii după un loc specificat;

c (change) — schimbă liniile specificate cu altele indicate;

d (delete) — șterge liniile specificate;

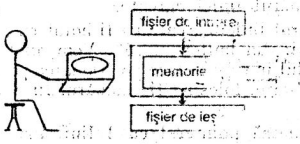


Fig. 2.9. Editarea programelor.

e (edit) — încarcă în bufferul de editare din memorie un anumit fişier;

f (filename) — tipăreşte, pentru aducere aminte, numele fişierului care se editează;

g (global) — aplică comenzile care urmează la întreg bufferul;

i (insert) — inserează liniile înainte de locul specificat;

j (join) — concatenează două linii în una;

m (move) — mută linii de la un loc în altul (cu ştergerea celor iniţiale)

n (number) — afişează liniile şi numărul lor de ordine;

p (pointer) — poziţionează pointerul liniei curente în buffer şi afişează liniile;

q (quit) — ieşire din editare;

r (read) — citeşte un fişier în buffer;

s (substitute) — înlocuieşte un şir de caractere cu un altul;

t (transfer) — copiază liniile, fără distrugerea celor originale;

x — intrare în mod cifrare;

w (write) — scrie bufferul pe fişierul de ieşire;

W (write) — adaugă bufferul la conţinutul fişierului de ieşire;

= — afişează valorea curentă a lui. (linia curentă) sau a lui $ (ultima linie).

* Pentru a putea adresa informaţia, editorul numerotează liniile din bufferul de editare, în secvenţă; prima linie primeşte numărul implicit 1, a doua 2 etc. Numărul de linie nu apare explicit în fişierul de editat. Editorul menţine un pointer care indică numărul liniei curente.

Prin convenţie numărul liniei curente va fi notat cu „.„ iar numărul ultimei linii din buffer cu „$". Vom analiza prin cîteva exemple modul în care se face poziţionarea pointerului în bufferul de editare, folosind fie numărul liniilor, fie notaţiile simbolice:

$ — 3p — poziţionează pointerul cu 3 linii înainte de ultima linie;

2,5 p — listează liniile 2, 3, 4, 5 şi se poziţionează pe linia 5;

3 p — afişează linia 3, poziţionîndu-se pe ea;

−2, +2p — afişează 5 linii în jurul liniei curente, poziţionarea făcîndu-se la a doua linie după linia curentă;

/sir/p — se poziţionează pe prima linie care conţine „şir" (căutarea se face în buffer circular în jos;)

/sir/p — poziţionarea se face la prima linie care conţine „şir" căutată în buffer circular în sus;

/şir/, $-1p — se listează de la prima linie ce conţine „şir" pînă la penultima, poziţionarea pointerului făcîndu-se pe aceasta din urmă;

\+ — avansează pointerul cu o linie;

++ — avansează pointerul cu 2 linii;

— — — — se face mutarea pointerului cu 3 linii în sus

Cîteva exemple vor arăta în continuare **modul de utilizare a unor comenzi** mai frecvent folosite;

$d — va face ştergerea, începînd cu linia curentă, a tuturor liniilor pînă la sfîrşitul bufferului;

1, $-1s/şir1/şir2 — va înlocui în tot bufferul „şir1" cu „şir2";

1, 3t $ — va transfera liniile 1, 2, 3 la sfîrşitul bufferului;

3a — va insera un număr de linii după linia 3; editorul intră în modul inserare pînă ce se tastează o linie cu „.". în prima coloană;

3i — va insera un număr de linii înainte de linia 3; editorul intră în mod inserare pînă la o linie cu „.".

* Editorul **ed** poate lucra şi în mod criptografic. Intrarea în acest mod se face cu **comanda x.**

În acest caz, la sfîrșitul editării, comanda **w**, va face scrierea pe disc a conținutului bufferului, nu în clar, ci sub forma cifrată:

```
$ed fis     — se intră în mod cifrare
x
key:        — se furnizează cheia
.           — se editează fișierul
.           — se cifrează conținutul bufferului și se scrie
w             pe disc
q
```

Pentru a putea edita un fișier cifrat se procedează astfel:

```
$ed         — se intră în mod cifrare
x
key:        — se furnizează cheia (aceeași ca cea de la
              scrierea fișierului
r fis       — se citește fișierul în buffer, făcîndu-se și
              descifrarea lui
.           — se editează fișierul
.
w           — se cifrează din nou conținutul bufferului
              și se scrie pe disc
```

În încheiere, vom prezenta un scurt exemplu de creare a unui fișier tex cu ajutorul lui **ed**:

```
$ed fis1    — se lansează editorul
 fis 1      — ed semnalează că fis1 nu există pe disc
a           — se intră în mod inserare
linia 1
linia 2     — se tastează conținutul fișierului text
linia 3
.           — se iese din mod inserare
1, $p       — se afișează conținutul bufferului
linia 1
linia 2
linia 3
w           — se scrie bufferul în fișierul fis1
21          — ed indică numărul de caractere transferate
q           — se iese din ed
$
```


2.7.2. Editorul de texte vi

Editorul **vi**, produs la Universitatea Berkely, a fost proiectat pentru extinderea facilităților lui **ed**.

Acesta afișa textul, dar nu și poziția pointerului de linie.

Editorul **vi** lucrează **în mod ecran**, marcînd în permanență linia curentă. Este un editor foarte complex care are peste 100 de comenzi, imens de puternice dar și mai greu de inventat cu toate facilitățile sale.

Vom prezenta doar cîteva din principalele sale funcțiuni.

Ideea de bază a funcționării lui este aceea că fișierul de editat este copiat într-un buffer de editare. Dar, spre deosebire de **ed**, ecranul devine o fereastră într-un text.

Dimensiunea ferestrei este variabilă și poate fi resetată de utilizator. Implicit, fereastra este jumătate de ecran la terminalele lente sau întreg ecranul la cele rapide.

* O serie de taste permit mutarea ferestrei în bufferul text (operație cunoscută sub numele de **scroll**):

CTRL/F — înainte o fereastră (*forword*);
CTRL/B — înapoi o fereastră (*backward*);
CTRL/D — înainte o parte din fereastră (*down*);
CTRL/U — înapoi o parte din fereastră (*up*).

Fiecare comandă de scroll CTRL/D și CTRL/U poate fi prefixată de un număr care reprezintă numărul de linii care sînt deplasate.

Acest număr nu numai că are efect asupra comenzii actuale, dar este reținut și pentru comenzile următoare. În cazul comenzilor CTRL/F și CTRL/B numărul care le prefixează are semnificația de stabilire a dimensiunii ferestrei.

* Mutarea cursorului de editare în cadrul ferestrei reprezintă o altă facilitate importantă a lui **vi**. Cursorul apare de obicei fie pe video invers, fie pe sclipire, marcînd poziția pe care se execută o anumită comandă.

Dintre numeroasele posibilități de deplasare a cursorului, vom prezenta doar pe cele mai folosite:

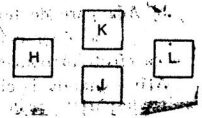


Fig. 2.10. Deplasarea cursorului între linii.

a) Deplasarea cursorului între linii se face cu 4 chei de la mijlocul tastaturii: **K, L, J, H** (fig. 2.10):

semnificație:

K — în sus;
L — la dreapta;
J — în jos;
H — la stînga.

b) Deplasarea cursorului în linie se face cu tastele:

w — înainte cuvînt;
b — înapoi cuvînt.

Cuvîntul este o unitate componentă a liniei, fiind separat de două blancuri.

c) Deplasarea cursorului la distanțe mari se face cu tastele:

H — la început de ecran (fereastra);
L — la ultima linie a ferestrei;
M — la mijlocul ferestrei.

Aceste comenzi pot fi prefixate cu un număr care reprezintă linii.

De exemplu, **5H** face deplasarea la a 5-a linie după început iar **3L** deplasează cursorul pe a 3-a linie de la sfîrșitul ferestrei.

* Comenzile editorului se tastează doar cînd acesta este în starea de „întrebare". Cele mai multe comenzi apar în partea de jos a ecranului, după ce sînt tastate. În general, ele au un **verb** format dintr-o singură literă, împreună cu unele argumente.

Tasta ESC se folosește pentru a indica sfîrșitul unei comenzi și a cere execuția sa. Iată cîteva din comenzile uzuale:

a) **Inserare de text** (*i*):

i ⟨ șir caractere ⟩ ESC

are ca efect inserarea șirului indicat înainte de cursor.

b) **Adăugare de text** (*a*):

a ⟨ șir caractere ⟩ ESC

are ca efect introducerea șirului indicat după cursor.

Atît comanda **i**, cît și a pot insera (adăuga) mai multe linii, despărțite prin RETURN (*new line*).

Sfîrșitul inserării (adăugării) este recunoscut prin caracterul ESC.

c) **Ștergere text** (*d*):

d ⟨ unitate ⟩

are semnificația că șterge una sau mai multe caractere, cuvinte sau linii prin ⟨ unitate ⟩:

dnw — semnifică ștergerea a **n** cuvinte înainte (după cursor);
dnb — semnifică ștergerea a **n** cuvinte înapoi (înaintea cursorului);
dnh — șterge **n** caractere în stînga cursorului;
dnl — șterge **n** caractere de la cursor în dreapta;
dnd — șterge **n** linii în jos (inclusiv linia curentă);
dnj — șterge **n** linii după linia curentă, plus linia curentă.

Iată și cîteva exemple:

d3w — permite ștergerea următoarelor 3 cuvinte;
dd — șterge linia curentă;
d1 — șterge caracterul indicat de cursor;
d51 — șterge 5 caractere de la cursor la dreapta.

d) **Schimbare text** (*c*):

c ⟨ unitate ⟩ ⟨ șir nou ⟩ ESC

are semnificația de ștergere a unității indicate și de înlocuire a ei cu șirul nou indicat. De exemplu:

cw noi ESC — va șterge următorul cuvînt din dreapta cursorului și îl va înlocui cu șirul „noi";
cc eu ESC — va șterge linia curentă și o înlocuiește cu șirul „eu".

* **Lansarea editorului** *vi* se face cu comanda:

$vi ⟨ fișier ⟩

unde ⟨ fișier ⟩ reprezintă numele fișierului ce se dorește a se edita.

* Ieșirea din editor 'D' Y , cu salvarea bufferului de editare în fișierul original se face cu comanda:

:wq ESC după tastarea prealabilă a lui CTRL/Z.

Ieșirea doar cu q face ca bufferul să fie șters fără să se facă copierea lui în fișier.

2.7.3. Limbajul C

Limbajul C este cel mai important limbaj al sistemului UNIX, în primul rînd datorită faptului că circa 80% din acest sistem de operare este scris în C.

Este primul limbaj care îmbină facilitățile de programare structurată, la nivel înalt (de tip PASCAL sau ALGOL) cu puterea și caracteristicile limbajelor de asamblare. În acest context multă lume consideră C ca un limbaj ideal de programare a sistemelor.

Limbajul C a fost inventat în 1973 de Denis Ritchie la firma AT&T, Bell Labs, pentru a furniza un limbaj de nivel înalt pentru scrierea sistemului UNIX. Între timp a devenit un limbaj popular, permițînd dezvoltarea unor programe simple și concise.

În continuare, vom face o sinteză a principalelor caracteristici ale acestui limbaj; pentru detalii recomandăm lucrarea fundamentală a lui Kernigham și Ritchie.

2.7.3.1. *Caracteristici ale limbajului C*

Structura limbajului de programare C îl plasează între FORTRAN și PASCAL, dar cu diferențe semnificative față de acestea în ceea ce privește un număr de operații dependente de mașină, pe care cele două limbaje le ignoră.

În acest context, dacă limbajul C este independent de calculator, programele C pot fi dependente adesea de structura hard pe care au fost scrise. Cîteva din trăsăturile sale cele mai importante sînt următoarele:

— este un limbaj mic, relativ ușor de compilat pe toată gama de calculatoare (micro, mini și medii-mari);

— are un set de instrucțiuni structurale, foarte puternice;

— expresiile care se pot construi sînt puternice, incluzînd drept caz particular atribuirea și apelul de funcții;

— introduce noi operatori care corespund unor instrucțiuni mașină sau unor moduri de adresare;

— permite definirea unor variabile pointer — cu adrese ale altor variabile — și folosirea adresării indirecte, specifice limbajelor de asamblare;

— posedă un set de funcții standard de I/O, care acoperă o largă diversitate în ceea ce privește lucrul cu fișiere (limbajul propriu-zis nu are instrucțiuni de I/O de tip READ/WRITE);

— permite utilizarea unor directive de includere de fișiere text, de macrosubstituție și compilare condiționată, întîlnite de asemenea în limbajele de asamblare;

— are un singur tip de procedură — funcția; transmiterea parametrilor la funcție se face prin valoare. Însă o serie de argumente pot fi adrese, ceea ce permite returnarea unor rezultate nu numai prin numele funcției.

Pentru a întări afirmația potrivit căreia C este un limbaj foarte puternic, vom face cîteva comparații între acest limbaj și PASCAL, pe de o parte, și limbajele de asamblare, pe de altă parte.

Comparație PASCAL — C

— au o serie de trăsături comune, cum ar fi: structuri de date, instrucțiuni de control structurate, necesitatea declarării tuturor variabilelor folosite, instrucțiuni compuse;

— în PASCAL sînt reguli stricte de concordanță între tipul operanzilor și operatorii folosiți. În C există o mai mare libertate în utilizarea variabilelor, ceea ce îi conferă acestuia o mai mare putere. Însă, deoarece compilatorul face mai puține verificări, există și riscul nedetectării unor erori de programare;

— în PASCAL se utilizează limitat variabilele de tip pointer, care sînt necesare doar atunci cînd se lucrează cu date alocate dinamic, în cursul execuției și ale căror adrese, nefiind cunoscute, nu pot fi referite prin variabile.

În C variabilele pointer au o utilizare largă: pot apărea în expresii, pot fi transmise ca argumente, pot fi folosite, în afară de variabilele cu alocare dinamică, la lucrul cu șiruri, tablouri sau fișiere;

— structura programelor PASCAL este monolit; un program include 1 ... *n* proceduri, care pot avea alte proceduri ș.a.m.d., pe mai multe niveluri. Programele C sînt structurate ca succesiuni de funcții, repartizate în unul sau mai multe fișiere sursă.

O funcție nu poate conține în interiorul ei definiția altor funcții; ca urmare, toate funcțiile programului sînt la același nivel;

— în PASCAL standard nu se recunoaște noțiunea de fișier extern.

Tratarea acestora depinde mult de implementare, ceea ce asigură mai puțină portabilitate respectivelor programe. În C fișierele de date joacă un rol important; ele sînt privite fie ca o succesiune de blocuri disc, fie ca o succesiune de articole logice, fie ca o succesiune de octeți. Accesul la ele poate fi secvențial sau direct, făcîndu-se prin funcțiile de bibliotecă, asemănător cu limbajele de asamblare (de ex., SGF). Ca urmare, există un set comun de funcții standard la toate implementările;

— există în C două stiluri de programare: unul tradițional, ca în PASCAL, și un altul care presupune experiență și care se bazează pe utilizarea intensivă a facilităților specifice C-ului (pointeri, operatori etc.), conducînd la programe compacte dar și mai greu de citit. Extensiile PASCAL pe mini și micro ca MT+ (sub CP/M), TURBO PASCAL (sub MS-DOS) sau PASCAL OREGON (sub RSX-11M) includ o serie de facilități specifice limbajului C.

Comparație C — Limbaje de asamblare

— C posedă o serie de facilități specifice limbajelor de asamblare: operatori de complementare, deplasare, incrementare, decrementare, obținerea adresei de memorie, operațiuni logice bit cu bit, adresare indirectă, alocare de registre pentru variabile.

Din aceste motive unii numesc C asamblor independent de mașină;

— codul rezultat în urma compilării programelor C este mult diferit decît cel obținut prin limbaj de asamblare, în special datorită folosirii memorării tuturor variabilelor în stivă. Codul obiect obținut în C este de cîteva ori mai mare.

2.7.3.2. *Structura programelor C*

Un program C este o colecție de funcții, fiecare prelucrînd zero sau mai multe argumente. O funcție în C este comparabilă în formă și obiectiv cu o funcție din PASCAL sau FORTRAN. Fiecare funcție are nume și parametri (argumente). În corpul funcției se găsesc declarații și instrucțiuni, închise între acolade, { }.

Și programul principal este o funcție cu un nume special: **main**. Iată un scurt un program C:

```
main () /* comentariu */
    int i, j, k; /* declar 3 variabile întregi*/
                 i=10;      /* i=10 zecimal */
                 j=i+015  /* j=i+15 în octal*/
                 k=i+j+0×ff /* 0×ff este în
                              hexazecimal*/
```

Programul principal nu are argumente. Toate variabilele utilizate trebuie declarate. Orice instrucțiune se termină cu ; care este **terminator** (spre deosebire de PASCAL unde este separator).

Programul ilustrează și cîteva tipuri de constante (zecimale, octale, hexa).

2.7.3.3. *Tipuri de date de bază*

Limbajul C are un set de tipuri de date, fiecăruia fiindu-i alocată o zonă de memorie de lungime diferită, care depinde de lungimea cuvîntului calculator (16, 32 sau 36 de biți). Există 3 tipuri de date de bază:

1) **tipul caracter** — se indică cu cuvîntul cheie **char** și i se alocă 8 biți la calculatoarele din familia PDP-11;

2) **tipul întreg** — se indică prin **int** și i se alocă 16 biți pe PDP-11;

3) **tipul real** — indicat prin **float** pentru simpla precizie (32 de biți) sau **double** pentru dubla precizie (64 de biți).

La tipul **int** se pot atașa 3 **calificatori**:

short int (sau **short**) — cînd întregul este reprezentat pe 16 biți;

long int (sau **long**) — cînd întregul este reprezentat pe 32 biţi;

unsigned int (sau **unsigned**) — cînd întregul este reprezentat fără semn.

Se observă că nu există tipul boolean; ca urmare, expresiile logice vor avea rezultatul întreg 0 (echivalent lui **false**) sau diferit de zero (echivalentul lui **true**).

2.7.3.4. *Constante*

Se pot folosi 4 tipuri de constante în programele C:

1) Constante întregi, care se pot exprima în 3 baze:
— zecimal (de exemplu, 131);
— octal, care începe cu zero (de exemplu, 0771);
— hexa, care începe cu Ox (de exemplu, Oxfa9).

2) Constante caracter, formate dintr-un singur caracter încadrat între apostrof, 'x'. Valoarea constantei este egală cu cadrul numeric al caracterului respectiv în ASCII sau EBCIDIC. Sînt cîteva caractere speciale precedate de '/':
/b — BS
/n — NL
/r — CR
/f — FF
/t — TAB
/o — NULL
/ddd-cod octal. d=0 ,..., 7

3) Constante reale, care pot fi cu exponent (de exemplu, 12.1e-10 cu semnificaţia 12.1*10-10) sau fără exponent (de exemplu, 13.21).

4) Constante şir de caracter, formate din succesiuni de caractere cuprinse între ghilimele. În reprezentarea lor în memorie, la sfîrşitul şirului se pune caracterul NULL (/O).

De exemplu, şirul „unix" este o constantă de tip şir.

2.7.3.5. *Variabile*

Variabilele desemnează prin numele lor locaţii ale memoriei interne. Se acceptă pentru numele variabilelor litere şi cifre, primul caracter fiind literă. Compilatorul reţine doar primele 8 caractere ale numelui.

Variabilele pot conţine valori sau adrese de memorie.

Variabilele care conţin adresele altor variabile se numesc variabile pointer. Operatorul & asociat la un nume de variabilă obţine adresa ei. De exemplu, p=& x face ca *p* să conţină adresa lui *x*. Ca urmare, există 2 moduri de referire a datelor în memorie:

— prin adresare directă (cu identificatorii variabilelor);
— prin adresare indirectă (folosind variabilele pointer).

Variabilele au 2 atribute în C:

a) tipul variabilei (**int, char, float, double** sau tipuri compuse);

b) clasa de memorare, care poate fi:

— **auto** — variabile alocate dinamic la execuţie, în stivă, în momentul intrării în blocul de program unde sînt definite şi dispar în stivă, la ieşirea din bloc; ca urmare, nu se poate conta pe valoarea lor la o nouă revenire în acel bloc de program;

— **static** — variabile alocate static la compilare şi care îşi păstrează valoarea pe toată durata de execuţie a programului;

— **extern** — variabile alocate static şi care au ca domeniu de valabilitate tot programul; sînt variabile globael, folosite la comunicarea între funcţii;

— **register** — variabile de tip **auto**, dar a căror stocare se face în registru (dacă este posibil).

Implicit, orice variabilă declarată în corpul unei funcţii se consideră de tip **auto**, iar cele declarate în afara funcţiilor, de tip **extern.**

De exemplu, se pot folosi declaraţii ca:

long int a, b, c;
double r, p;
register float al;
static u, w;
char c, bs = '//'.

Toate variabilele folosite trebuie declarate, cu sau fără iniţializare.

2.7.3.6. *Expresii*

Expresiile sînt formate din operanzi şi operatori. Pe post de operanzi pot apare valori şi adrese. Valorile se indică prin variabile sau constante. Operatorii aparţin uneia din următoarele clase:

a) **operatori aritmetici:**

+	adunare;
−	scădere;
×	înmulţire;
/	împărţire;
%	reducere modulo;

b. **operatori relaţionali:**

>	mai mare;
<	mai mic;
>=	mai mare sau egal;
<=	mai mic sau egal;
==	egal;
!=	diferit;

c. **operatori logici între expresii:**

&&	SI logic;
||	SAU logic;

d. **operatori logici între şiruri de biţi:**

&	ŞI logic;
:	SAU inclusiv;
^	SAU exclusiv (suma modulo doi);
<<	deplasare stînga;
>>	deplasare dreapta;
~	complement de 1;

e. **operatori de incrementare/decrementare:**

++	adună 1 la operand;
−−	scade 1 din operand;

f. **operatori de asignare:**

=	asignare;
+=, −=, *=, /=, %=	operatori de asignare.

Operatorii de asignare înlocuiesc expresiile de forma:

⟨variabilă⟩= ⟨variabilă⟩ ⟨operator⟩ ⟨expresie⟩

cu forma mai scurtă:

⟨variabilă⟩ ⟨operator⟩= ⟨expresie⟩

care precede şi un cod mai eficient în urma traducerii.

Exemple de expresii:

a ⟨max && (b − 1)!=0 (se verifică un şir de condiţii)

++i (face $i = i + 1$)

−−i (face $i = i - 1$)

b << 2 (face deplasarea lui b cu două poziţii la stînga)

i += 2

(face i = i + 2)

2.7.3.7. *Tipuri de date complexe*

În această categorie intră patru clase de date complexe: tablourile, structurile, reuniunile şi pointerii.

Tablourile sînt colecţii de date omogene (de acelaşi tip).

Un tablou poate fi unidimensional (vector), bidimensional (matrice) sau cu trei dimensiuni. La declararea lui se indică numărul maxim de elemente. De exemplu:

```
int a [10] ;
```

declară un vector de 10 întregi, referiţi cu *a* [0] ,..., *a* [9];

```
float b [10] [20] ;
```

declară o matrice de 10 linii şi 20 de coloane.

În funcţii se poate declara şi un tablou fără dimensiuni (de exemplu, a[]), caz în care dimensionarea tabloului se va face la apelul funcţiei, condiţionat de numărul de elemente sosite ca parametri.

Structuri. O structură reprezintă o colecţie de variabile, în mod uzual de tipuri diferite (neomogene), grupate împreună sub un singur nume. De exemplu, declaraţia:

```
struct data {
        int zi;
        int luna;
        int an;
        char nume — luna [4];
```

definește o structură de patru membri. Pentru a declara variabile care să fie conforme cu aceste structuri, numele lor poate fi pus între } și ; sau se putea scrie ulterior instrucțiunea:

```
struct data d1, d2, d3;
```

Pentru a face asignarea unor valori la elementele unei variabile structurate vom proceda de exemplu:

```
d1. zi =31;
```

unde *d1* este numele variabilei, iar *zi* este numele cîmpului din structură.

O structură poate referi alte structuri, ca în exemplul următor:

```
struct persoană {
        char nume [15];
        char prenume [15];
        long salariu;
        struct data zi-naștere;
        struct zi-angajare;
        }  dan, serban;
```

După cum se poate observa, cîmpurile *zi-naștere* și *zi-angajare* sînt structuri de tip *data*, declarate mai sus. Putem referi cîmpuri ale variabilelor de forma *dan. salariu* sau *serban. zi-naștere. an.*

Una din structurile de date mult folosite în programe sînt tabelele, care nu sînt altceva decît matrice de structuri.

De exemplu:

```
struct persoană tabel-studenți [100];
```

este o tabelă ale cărei linii conțin datele de tip *persoană*.

În fine, se pot declara structuri ale căror cîmpuri să fie formate dintr-un număr dorit de biți, ca în exemplul:

```
struct{
        unsigned k1:3;
        unsigned k2:1;
        }fanion;
```

Variabila fanion are 2 cîmpuri: *k1* de 3 biți și *k1* de 1 bit.

Uniunile. O uniune este o colecție de membri care la un moment dat poate reține doar pe unul din ei. Declarația:

```
union {int i; char c} a;
```

definește variabila *a* care poate fi de tip întreg sau de tip caracter, dar nu amîndouă în același timp. Dacă este referită cu a.i se consideră întreg, iar cu a.c — caracter. În acest fel se pot manipula în aceeași zonă de memorie diferite tipuri de date.

Pointeri. Pointerii sînt folosiți pentru memorarea adreselor. În declararea variabilelor de tip pointer se folosește caracterul /*/ care prefixează numele variabilei. În exemplul:

```
int a, *p, *b [10] ;
```

se declară întregul *a*, variabila pointer *p* și matricea de 10 pointeri, *b*. Pentru a obține adresa unei variabile se folosește operatorul & :

```
p = & a;
```

unde *p* va conține după asignare adresa lui *a*. O asignare de forma:

```
q=*p;
```

nu va depune în *q* conținutul lui *p*, ci ceea ce se găsește la adresa din *p* (acces indirect).

2.7.3.8. *Instrucțiuni pentru controlul fluxului*

Fiecare instrucțiune se termină cu caracterul ;

Un set de instrucțiuni încadrate de { } alcătuiesc un **bloc** sau o **instrucțiune compusă**, care din punct de vedere sintactic este echivalentă cu o instrucțiune simplă.

Deciziile se codifică prin 3 instrucțiuni:

- **IF-ELSE** care are sintaxa:

```
if (expresie)
        instrucțiune-1
else
        instrucțiune-2
```

Se execută instrucțiune-1 dacă expresia este adevărată și instrucțiune-2 în caz contrar.

- **ELSE-IF** pentru decizii multiple:

```
if (expresie-1)
        instrucțiune-1
else if (expresie-2)
        instrucțiune-2
else
        instrucțiune-n
```

Se va executa instrucțiune-1 dacă expresie-1 este adevărată, instrucțiune-2 dacă expresie—2 este adevărată s.a.m.d.

• **SWITCH** pentru decizii multiple:

```
switch (expresie) {
        case v1:
                    instrucțiune—1
        case v2:
                    instrucțiune—2
        .
        .
        .
        case vn:
                    instrncțiune—n
        default
        }           instrucțiune n+1
```

Se va evalua mai întîi expresia. Dacă rezultatul are valoarea *vi* se va executa instrucțiune *i*, instrucțiune *i*+1, pînă la sfîrșit sau pînă la instrucțiunea **break**.

Ciclurile se pot codifica cu:

• **WHILE** pentru ciclul cu test inițial:

```
while (expresie)
            instrucțiune
```

Se va executa instrucțiune cît timp expresia este adevărată (diferită de 0).

• **DO-WHILE** pentru ciclul cu test final:

```
do instrucțiune
while (expresie)
```

Se va executa instrucțiune cît timp expresia este adevărată (diferită de 0).

• **FOR** pentru ciclul cu contor:

```
for (expresie—1; expresie—2; expresie—3)
            instrucțiune
```

Se va executa instrucțiune, pentru valori ale contorului care pleacă de la expresie—1, pînă ce se îndeplinește condiția expresie—2. La fiecare nou ciclu se modifică conform lui expresie—3.

Ieșirile din cicluri se pot face și cu instrucțiunile:

break — care permite ieșirea dintr-un ciclu **do, for, while, switch** înainte de terminarea lui;

continue — care permite trecerea la următoarea iterație a ciclului, înainte de terminarea celei în curs de execuție.

2.7.3.9. *Funcții*

După cum s-a arătat, un program C reprezintă o colecție de funcții, unele declarate de utilizator, altele din bibliotecă.

Structura unei funcții este:

```
[tip] nume (lista—argumente)
declarare—argumente
{
corp-funcție
}
```

Tipul implicit al funcțiilor este **int**. Argumentele funcției (dacă există) se declară înainte de corpul ei. Revenirea dintr-o funcție se face fie cu instrucțiunea **return**, fie implicit la sfîrșitul funcției. Argumentele sînt transmise prin valoare, și nu prin adresă, ceea ce înseamnă că orice modificări ale lor în corpul funcției nu afectează valorile lor din funcția apelantă. Să mai observăm că funcțiile se pot apela recursiv.

2.7.3.10. *Preprocesorul C*

Înainte ca un program să fie compilat, el este automat prelucrat de un preprocesor care face 3 transformări ale textului sursă:

1. Include fișierele specificate cu directiva **include**:

```
#include „fisier.h"
```

Se mai folosește forma:

```
#include ⟨fisier.h ⟩
```

în care căutarea se face în directorul special „/usr/include" și nu în directorul curent ca în prima formă.

2. Înlocuiește macrodefinițiile, date cu instrucțiunea **define**:

```
#define nume valoare
```

În textul sursă se va înlocui nume în valoarea declarată.

2.7.3.11. *Intrări/ieşiri*

Facilităţile de I/O nu fac parte din limbajul C, acesta neposedînd instrucţiuni destinate acestui scop. Dar există o bibliotecă standard din care se pot apela funcţii care să realizeze aceste operaţii. Accesul la biblioteca standard trebuie să fie precedat de o instrucţiune:

#include ⟨stdio.h⟩

care va permite înlocuirea în programul sursă a unor macrodefiniţii şi variabile utilizate de funcţiile bibliotecii. Principalele funcţii care servesc accesului la funcţiile standard sînt:

getchar — cu funcţia de citire a caracterului următor pe fişierul standard de intrare;

putchar — cu funcţia de scriere a caracterului pe fişierul standard de ieşire;

printf — cu funcţia de conversie a unor valori numerice şi de tipărire a lor pe fişierul standard de ieşire;

scanf — cu funcţia de citire a unor valori de pe fişierul standard de intrare şi de conversie a lor conform unei liste de formate.

Există însă un număr mare de alte funcţii care se pot apela şi care permit manipularea fişierelor disc în acces secvenţial şi direct, generarea dinamică de procese, controlul execuţiei proceselor, alocarea memoriei, controlul directoarelor etc. şi care conferă o putere deosebită limbajului C.

2.7.3.12. *Compilatorul C*

Compilatorul limbajului C sub UNIX este catalogat sub numele **cc**. La fel ca şi compilatorul f77, **cc** se produce la ieşire cod intermediar C. Acest cod este translatat în limbaj de asamblare de un alt program care, este denumit „pasul 2 al compilatorului C" (/lib./c 2.c). Codul intermediar este independent de implementare. Textul obţinut după pasul 2 este translatat în final în cod obiect relocabil, din care se va putea genera cod executabil (fig. 2.11).

În momentul lansării compilatorului **cc**, se creează automat un pipe între mai multe programe corespunzătoare pasului 1, pasului 2, asamblorului şi editorului de legături.

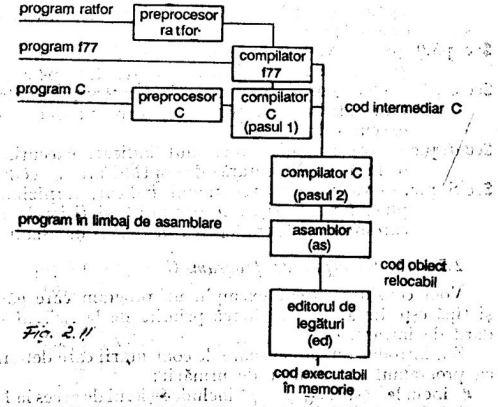


Fig. 2.11. Etapele intermediare ale traducerii programelor.

Lansarea compilatorului se face în mod uzual cu comanda:

$cc ⟨ fişier-program ⟩

unde ⟨ fişier-program ⟩ este numele fişierului sursă cu programul C; el va purta de obicei sufixul „c". Programul cod obiect executabil va fi depus în fişierul cu nume standard „a.out", ceea ce înseamnă că ultimul proces din pipe este editorul de legături **ld**.

De asemenea, în urma compilării vom folosi opţiunea **—o**:

$cc —o ⟨ fişier-obiect ⟩ ⟨ fişier-program ⟩

Vom ilustra şi alte posibilităţi ale compilatorului, prin cîteva exemple:

$cc-P prog. c — va compila programul „prog. c" şi va furniza la ieşire prog. i (ieşirea preprocesorului) datorită opţiunii —P, „prog. o" şi „a.out";

$cc p1.c p2.c — va executa compilarea separată a două programe, reprezentînd cele 2 fişiere obiect relocabile

„p1.o" și „p2.o", precum și fișierul executabil „a.out";

$cc p3.0 p4.c — va combina modulele obiect din „p1.o" și ceea ce rezultă din traducerea lui „p4.c";

$cc-c p1.c — va compila programul indicat, suprimînd însă faza editării legăturii (*ld*) care produce codul executabil;

$cc-0 p.c. — va compila programul indicat, executînd și o fază suplimentară de optimizare a codului;

$cc-S p.c. — va compila programul indicat, furnizînd, în plus, față de fișierele obișnuite și fișierul „p.s." care reprezintă traducerea în limbaj de asamblare.

2.7.3.13. *Exemplu de program C*

Vom considera drept exemplu un program care găsește și tipărește linia cea mai lungă primită de la fișierul standard de intrare.

S-a introdus un număr mare de comentarii care determină ca programul să fie ușor de urmărit:

```
# include ⟨stdio.g ⟩    /* include fișierul de acces la I/E*/
# define EOF '$'        /* $indică sfîrșit de fișier */
# define MAXLINE 80     /* dimensiunea maximă a liniei */
  main ()               / * funcția principală */
  {
  int len;              /* lungimea unei linii */
  int max;              /* lungimea maximă a liniilor anterioare */
  char line  [MAXLINE]; /* linia curentă */
  char save  [MAXLINE]; /* linia cea mai lungă */
  max = 0;              /* aici începe programul */
  while (len = getline (line, MAXLINE)) ⟩ 0) {
    if (len ⟩ max)
      {
      max = len;
      copy (line, save);
      }
    if (max ⟩ 0)
      printf („LUNGIMEA MAXIMĂ: %d/n", max);
      printf („%s", save);
      }
```



```
}
getline (s, lin) /* funcția de citire linie și determinare lungime */
  char s[] ;      /* declarare parametri */
  int lim;
  {
     int, c, i;
     for (i = 0; i ⟨ lim − 1 && (c = getch()) != EOF;++i) {
        s [i] = c;
        s [i] = '/0';
        return(i);
        }
copy (s1, s2) /* funcție de copiere șir s1 în s2 */
  char s1[]  , s2[]  ;
  {
  int i = 0;
  while ((s2 [i]  = s1 [i] ! = '/−')
    ++i;
    }
```

2.7.4. **Pascal Berkely**

PASCAL Berkely reprezintă poate cea mai răspîndită versiune de compilator de limbaj PASCAL, realizat la Universitatea Berkely din California. În structura sa PASCAL Berkely interpretează 6 comenzi: **pi, px, pix, pc, pxp, și pxref** care reprezintă apelări fie la interpretarea programului, fie la compilarea lui efectivă cu generarea de cod. Din punctul de vedere al sintacticii și semanticii este compatibil cu versiunea de PASCAL standardizată de ISO. Vom analiza pe scurt rolul fiecăreia din cele 6 componente ale sistemului:

• **pi** este un translator de tip **interpretor** care, plecînd de la textul sursă PASCAL, va genera cod interpretativ (intermediar).

Lansarea programului se face cu comanda:

$pi ⟨ opțiune ⟩ ⟨ fișier ⟩

Fișierul reprezintă textul sursă PASCAL, numele său trebuind să aibă extensia „.p" (de exemplu, prog. p).

În urma traducerii, codul intermediar obținut va fi depus într-un fișier cu nume standard: **obj**.

Principalele opțiuni ale lui **pi** sînt:

— *i* — se listează fișierele incluse în text cu directiva # include (similar cu limbajul C);
— *l* — se listează programul sursă;
— *p* — listează „post mortem" o eroare;
— *s* — se testează doar PASCAL standard (fără extensii);
— *t* — suprimă verificarea depășirii limitelor;
— *z* — se inițializează contoare pentru profile (cu **pxp**);

• **px** reprezintă monitorul de execuție al programelor PASCAL, traduse de **pi**. Ca urmare, **px** va interpreta codul intermediar din fișierul obj, generînd instrucțiuni mașină pe care le execută.

Lansarea se face cu comanda:

$px

• **pix** permite apelarea simultană a lui **pi** și a lui **px**, care vor comunica prin pipe. În acest fel se realizează traducerea și interpretarea programelor PASCAL.

Există o limită maximă pentru fișierele sursă ce sînt interpretate: 2 000 de linii. Lansarea:

&pix ⟨ fișier ⟩

• **pc** reprezintă compilatorul de limbaj PASCAL.

El poate genera la ieșire fie cod executabil, fie programa în limbaj de asamblare. Aceste module pot fi legate cu alte programe, scrise în alte limbaje. Înainte de a compila un program cu ajutorul lui **pc** este indicată punerea lui la punct cu **pi**, care face o bună localizare a erorilor. Lansarea compilatorului se face astfel:

$pc ⟨ opțiune ⟩ ⟨ fișier ⟩

Principalele opțiuni disponibile sînt:

— b ⟨n ⟩ — stabilește dimensiunea bufferului de ieșire la ⟨n ⟩;
— *c* — compilează un program parțial (pentru compilare incrementală);
— *i* — listează fișierele incluse cu # include;
— *o* ⟨fișier ⟩ — indică numele fișierului de ieșire (implicit a.out);
— *O* — optimizează codul generat;

— *s* — verifică PASCAL standard;
— *S* — ieșire în limbaj de asamblare;
— *z* — contoare pentru *pxp*;

• **pxp** este un profiler de execuție, specificînd de cîte ori este executată fiecare instrucțiune. Poate fi folosit la punerea la punct a programelor sau pentru detectarea acelor zone din program care sînt mai des folosite. De asemenea, poate fi utilizat pentru indentarea programului („beautifier").

Lansarea se face cu comanda:

$pxp ⟨opțiune ⟩ ⟨fișier ⟩

Din opțiunile de profilare amintim:

— *a* — se includ în profilare toate rutinele;
— *t* — tabelează opțiunile de proceduri și funcții.

Principalele opțiuni de formare a textului sursă sînt:

— *d* — suprimă listarea declarațiilor;
— *j* — face indentarea la stînga;
— *s* — comentariile se listează parțial;
— - — se subliniază cuvintele cheie ale programului;
— ⟨*nr* ⟩ — se stabilește valoarea de indentare (implicit 4).

Pentru a vedea de cîte ori s-au executat diferitele instrucțiuni ale programului, îl vom executa cu comanda:

$pix-z ⟨program ⟩

și apoi vom vizualiza contoarele cumulative astfel:

$pxp-z ⟨program ⟩

• **pxref** este un generator de liste de referințe încrucișate, indicînd modul de utilizare a variabilelor în programele mari.

2.7.5. FORTRAN 77

În sistemul de operare UNIX, compilatorul FORTRAN f 77 acceptă cea mai recentă versiune a acestui limbaj, FORTRAN 77. Aceasta reprezintă un nou standard, ulterior celui din 1966, dar programele FORTRAN 66 pot fi făcute executabile sub f77 fără mari dificultăți.

Deoarece FORTRAN este un limbaj foarte bine cunoscut, cu o literatură bogată, nu vom insista pe descrierea lui, ci doar pe aspectele caracteristice ale implementării sub UNIX.

Compilatorul f77 nu translatează programele FORTRAN direct în cod obiect sau în limbaj de asamblare. El produce mai întîi cod intermediar C, lucru care implică o serie de facilităţi suplimentare, specifice limbajului C, cum ar fi apelare recursivă a subrutinelor, disponibilitatea tuturor subrutinelor şi apelurilor sistem accesibile din C, intermixarea unor porţiuni de program scrise în FORTRAN şi C (apelului de funcţii, proceduri dintr-un limbaj în celălalt).

Aceste facilităţi fac ca versiunea f77 să se depărteze, în mică măsură însă, faţă de standardul FORTRAN 77.

* *Lansarea procesului de compilare* se face cu comanda:

$ f77 ⟨ program ⟩

unde ⟨ program ⟩ este fişierul sursă cu programul FORTRAN.

Terminarea compilării este marcată de apariţia prompter-ului „$". Dacă programul nu a avut erori sintactice, se generează codul obiect în fişierul „**a.out**", din directorul curent.

Se poate însă, prin opţiunea **—o**, să atribuim alt nume fişierului obiect:

$ f77-o ⟨ nume ⟩ ⟨ program ⟩

Există o serie de opţiuni ale compilatorului, dintre care interesante sînt cele legate de depanarea programului:

— *u* — face ca toate variabilele să fie nedefinite, forţînd utilizatorul să declare explicit orice variabilă pe care o foloseşte (ca în PASCAL, C);

— *w* — se suprimă mesajele de atenţionare;

— *c* — este o opţiune foarte folosită în aplicaţiile matematice, făcînd verificarea limitelor (indicilor) la fiecare utilizare a unui tablou.

* **Extensii lexicale**

La fel ca la vechile versiuni, textul sursă FORTRAN 77 este introdus pe linii, fiecare linie fiind completată în

intervalul coloanelor 1-72. Primele 5 coloane conţin eticheta, coloana 6 indică continuarea, iar în intervalul 7-72 se introduc instrucţiunile. Compilatorul f77 este însă mai tolerant cu privire la aceste şabloane. Liniile pot fi tastate în orice zonă a spaţiului disponibil iar liniile de continuare trebuie să conţină caracterul "$" pe prima coloană. Eticheta trebuie despărţită de instrucţiune printr-un caracter **tab**. Compilatorul f77 recunoaşte atît caracterele alfabetice mici, cît şi pe cele mari, pe care le translatează în mici (această translatare poate fi inhibată).

Cuvintele cheie sînt recunoscute însă doar sub forma literelor mici (este corect *a* = sin(*b*) sau *A* = sin(*b*), dar incorect *a* = SIN (*b*).

Constantele şir de caracter se pot da fie sub forma **nh** ⟨ **text** ⟩ (forma Hollerith), fie între apostroafe: „**text**".

Din motive de compatibilitate cu C, f77 recunoaşte caracterele precedate de „/":

/*n* — new line;
/*t* — tab;
/*b* — backspace;
/*0* — caracterul null;
/*f* — form feed.

* **Extensii sintactice**

Compilatorul f77 recunoaşte două tipuri de date care nu există în standard:

Declaraţiile:

integer * 2 i, j, k

double complex a, b, c

dublează dimensiunile de memorare a unor constante întregi sau complexe. În mod adecvat şi funcţiile intrinseci au fost îmbunătăţite, astfel încît să poată manipula variabile de acest tip. Declaraţia:

implicit undefined (a-z)

are rolul de a invalida orice tratare implicită a variabilelor; ca urmare, va fi necesară declararea explicită a tuturor variabilelor. Pentru iniţializarea variabilelor, instrucţiunea **data** a fost extinsă pentru a putea declara constantele octale (**o**), hexa (**z**), sau binare (**b**). De exemplu:

data a, b, c/b'0110', o'176'.z'1f'/

La fel ca și în limbajul C, f77 permite folosirea instrucțiunii **include**:

include ⟨ fișier-text ⟩

cu rolul de a insera în locul ei conținutul unui fișier text cu instrucțiuni FORTRAN.

*** Intrări/ieșiri**

Operațiile de intrare/ieșire sînt conectate la unitățile logice de intrare/ieșire, identificate prin numere (între 0 și 9), dintre care trei sînt asignate implicit:

— unitatea 5 — la fișierul standard de intrare (tastatură);
— unitatea 6 — la fișierul standard de ieșire (ecranul);
— unitatea 0 — la fișierul standard de erori (ecranul).

Celelalte șapte unități logice pot fi asignate unor fișiere secvențiale pe care le prelucrează fișierul, cu declarația **open**; de exemplu:

open (2, file = 'fișier')

Instrucțiunea **close**:

close (2)

va întrerupe asignarea făcută cu **open**.

Se pot declara fișiere atît în acces secvențial, cît și direct, instrucțiunea **open** avînd o serie de argumente care servesc acestui scop.

2.7.6. Ratfor

RATFOR-*RAT*ional *FOR*tran reprezintă o extensie a limbajului FORTRAN care include o serie de facilități specifice limbajelor structurate, ca C, PASCAL etc.

El îmbină calitățile limbajului FORTRAN în aplicațiile tehnico-științifice, marea sa răspîndire, cu avantajul programării structurate specifice limbajelor noi. RATFOR este un *preprocesor*, care face translatarea programelor în FORTRAN. Apoi acestea se vor compila cu f77 urmînd traseul cunoscut pînă la cod obiect executabil.

* *Lansarea preprocesorului RATFOR* se face în două moduri:

1. direct

$ ratfor ⟨ fișier ⟩

cînd se face translatarea programului în unul compatibil cu FORTRAN 77.

2. prin f77

$f77 (⟨ fișier.r ⟩

În cazul în care programul sursă furnizat lui f77 are extensia **.r**, compilatorul apelează automat RATFOR și apoi efectuează traducerea în fișierul obiect **a.out**.

***Extensii lexicale**

Instrucțiunile RATFOR pot fi scrise oriunde pe linie și continuate pe oricîte linii.

Preprocesorul va face ca instrucțiunile FORTRAN de la ieșire să înceapă în coloana 7 și să aibă caracterul de continuare în 6.

În general, la fel ca în C, se folosește caracterul „;" pentru separarea instrucțiunilor (care pot apărea mai multe pe aceeași linie).

Etichetele instrucțiunilor apar ca și în FORTRAN înaintea instrucțiunii propriu-zise. Totuși, programarea structurată în RATFOR face posibilă evitarea etichetării instrucțiunilor.

Operatorii aritmetici, logici și relaționali acceptați în RATFOR sînt identici cu cei din C (vezi § 2.7.3).

Există un caracter special „%", numit *operator de transparență* care, dacă este prezent la începutul unei linii RATFOR, va indica preprocesorului că linia respectivă va fi trecută în textul FORTRAN nemodificată.

* *Limbajul RATFOR* permite, ca și PASCAL și C, să se realizeze grupe de instrucțiuni. În acest scop se folosesc parantezele acolade {}, toate instrucțiunile cuprinse între ele alcătuind din punct de vedere sintactic o singură instrucțiune. De exemplu, instrucțiunea:

if(x == 0) {call mesajer; răspuns = 0; return}

folosește un astfel de grup de instrucțiuni.

* **Decizia** se codifică cu instrucțiunea **if-then-else** care are următoarea sintaxă:

if ⟨ condite ⟩ **then**
⟨ **instrucțiune** ⟩
else
⟨ **instrucțiune** ⟩

sau forma prescurtată:

if ⟨ condiție ⟩ **then** ⟨ **instrucțiune** ⟩

De exemplu:

if(x == 0) **then**

{a = b; j = 1}

else

{a = + b; j = 1}

* **Ciclul** se poate implementa cu una din cele 4 instrucțiuni:

1. **do** ⟨ var ⟩ = ⟨ n1 ⟩, ⟨ n2 ⟩{... ⟨ instrucțiune ⟩ ...}, luînd toate valorile întregi cuprinse între ⟨ n1 ⟩ și ⟨ n2 ⟩.

De exemplu:

do i = 1, n

{aux = a(i); a(i) =a(i + 1); a(i + 1) = aux}

2.

for (⟨ var ⟩ = ⟨ n1 ⟩; ⟨ condiție ⟩; ⟨ expresie ⟩) {... ⟨ instrucțiune ⟩...}

Se va executa ciclul cît timp condiția este îndeplinită; ea se referă de obicei la variabila de ciclu ⟨ var ⟩. De exemplu:

for (i = 1; i ⟨ n; i = i + 1)

{aux = a(i); a(i) = a(i + 1); a(i + 1) = **aux**}

3. **while** (⟨ condiție ⟩) {... ⟨ instrucțiune ⟩...}

Această instrucțiune reprezintă ciclul cu test inițial și repetă grupul de instrucțiuni **cît timp** ⟨ condiție ⟩ este îndeplinită. De exemplu:

while (x ⟩ 0)

{a = x; x = (b + c) * 2}

4. **repeat** {... ⟨ instrucțiune ⟩...}

until (⟨ condiție ⟩)

Această instrucțiune reprezintă ciclul cu test final și repetă grupul de instrucțiuni **pînă cînd** ⟨ condiție ⟩ devine adevărată.

De exemplu:

repeat (x ⟩ = 0)

{ a = x; x = (b + c) * 2}

La fel ca și în C, RATFOR furnizează două instrucțiuni suplimeștare, **next** și **breack**.

Next permite abandonarea iterației în curs și trecerea, dacă condiția permite, la iterația următoare. Instrucțiunea **breack** permite ieșirea din ciclu, înainte de terminarea sa.

* În mod similar cu C, 2 instrucțiuni permit substituția în textul sursă, **define** și **include**, cu semnificația cunoscută din paragraful 2.7.3.

În concluzie, RATFOR se dovedește un excelent mijloc de a programa în FORTRAN, cu structuri tipice limbajului C. El a fost preluat și implementat sub multe alte sisteme de operare.

* În UNIX există și un procesor numit **struct** care face operația inversă de generare a unor programe RATFOR plecînd de la sursa FORTRAN.

2.8. FACILITĂȚI DE PROTECȚIE CRIPTOGRAFICĂ LA UNIX

Un aspect interesant al acestui sistem de operare este și acela că, prin versiunile sale succesive, și-a perfecționat și îmbogățit facilitățile de protecție a informației și a mediului de execuție.

Ca o parte a ansamblului său standard de apeluri sisteme și comenzi, UNIX încorporează o serie de programe de protecție criptografică, care-l fac capabil să execute sarcini în medii „ostile". În cele ce urmează vom analiza principalele facilități criptografice ale acestui sistem de operare, căutînd să subliniem pentru fiecare atît aspectele pozitive, cît și deficiențele. Vom încerca să descriem eficiența diferitelor programe criptografice definind un coeficient criptografic, egal cu raportul dintre timpul cerut de un anumit program criptografic pentru a prelucra un fișier mare și timpul de transformare „nulă", adică simpla copiere, a aceluiași fișier. Prin timp vom înțelege timpul total, utilizator și sistem, și nu timpul real.

2.8.1. Facilități criptografice în versiunea 6

Versiunea 6 UNIX acceptă 2 pachete criptografice:

1. Pentru *cifrarea datelor* se furnizează un program care simulează software modelul mașinii M-209 sau CD-48 Hagelin Cryptograph utilizată de armata S.U.A. în ultimul război mondial.

Se folosea o cheie de 130 caractere, obținută prin expandarea unei chei scurte, introdusă de utilizator (formată din

orice caracter tipăribil). Cheia stabileşte poziţia „roţilor dinţate" care generează şirul cheie, însumat modulo cu textul clar.

2. Pentru *cifrarea parolelor* există o rutină de cifrare Hyper-Hagelin, în limbaj de asamblare, cu un număr mult mai mare de „roţi dinţate", deci cu o complexitate ridicată.

Algoritmul însă, cu o cheie dată, produce un cifru uşor de inversat; este mult mai dificil să se găsească cheia dacă se cunosc textul clar şi textul cifrat. De aici ideea de a se folosi parola introdusă drept cheie, cifrîndu-se o constantă şi punînd rezultatul în fişierul de parole. Din păcate, simularea lui M-209, în afara securităţii sale mediocre, este lentă, ducînd la obţinerea unui coeficient criptografic de 13,4 sau mai mare, chiar pe sisteme rapide.

2.8.2. Facilităţi criptografice în versiunea 7

Versiunea 7 a lui UNIX îmbogăţeşte setul de facilităţi criptografice, înlăturînd şi unele deficienţe semnalate în versiunea anterioară.

2.8.2.1. *Protecţia parolelor*

Cifrarea parolelor se face acum, utilizîndu-se simularea software a cifrului DES. Aceasta reprezintă primul standard de cifrare elaborat de NBS din SUA în 1979. DES cifrează blocuri de 64 de biţi, utilizînd o cheie de 56 de biţi. Algoritmul, folosit atît la cifrare cît şi la descifrare, este prezentat schematic în fig. 2.12. Blocul de intrare, T, este mai întîi permutat, obţinîndu-se $T_0 = \mathrm{IP}(T)$.

După aceea T_0 trece prin 16 iteraţii, funcţionînd identic, iar la sfîrşit este supus permutării inverse, IP-1. Cele 16 iteraţii ale funcţiei f combină substituţiile cu transpoziţiile. Fie T_i rezultatul iteraţiei i, iar L_i şi R_i părţile stîngă şi dreaptă, de cîte 32 biţi fiecare, ale lui T_i.

Atunci,

$$L_i = R_{i-1};$$

$$R_i = L_{i-1} \oplus f(R_{i-1}, K_i).$$

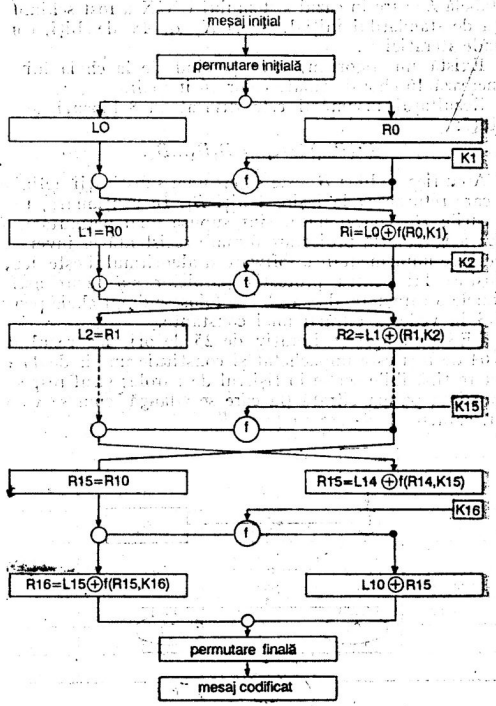


Fig. 2.12. **Algoritmul DES.**

Funcția f (fig. 2.13) începe prin a expanda intrarea (R_{i-1}) de 32 biți la $E(R_{i-1})$ de 48 de biți, utilizînd o tabelă E, care în cazul sistemului UNIX a fost schimbată față de standardul inițial. Cheia K_i, de 48 de biți, corespunde iterației i.

Există un algoritm, care plecînd de la cheia inițială generează 16 chei destinate celor 16 iterații.

Rezultatul însumării este divizat în 8 blocuri de cîte 6 biți:

$$E(R_{i-1}) \oplus K_i = B_1 B_2 ... B_8.$$

Apoi fiecare bloc B_i este supus unei substituții neliniare S_i care returnează la ieșire 4 biți B_i. În continuare, toate blocurile B_i, concatenate, sînt supuse unei permutări, P. DES asigură prin proiectare o mare rezistență la inversare. Deoarece implementarea software a algoritmului este lentă, folosirea lui pentru protecția parolelor s-a făcut astfel: primele 8 caractere ale parolei se folosesc drept cheie pentru DES în vederea cifrării unei constante.

DES este aplicat iterativ de 25 de ori, iar rezultatul de 64 de biți este împachetat și constituie un șir de 11 caractere tipăribile scrise în fișierul de parole; șirul respectiv reprezintă parola cifrată (la care se adaugă, cum se va vedea, caracterele — „sarea").

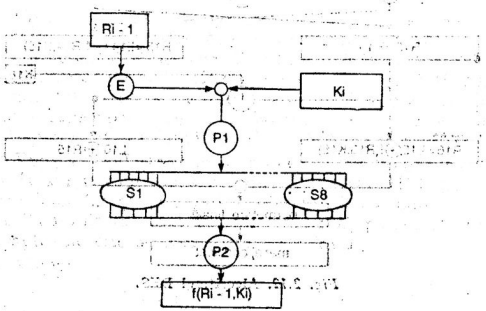


Fig. 2.13. Calculul funcției **F**.

Există 2 căi posibile de atac asupra schemei de cifrare. O posibilitate o reprezintă găsirea unei metode generale de inversare a algoritmului de cifrare, care ar permite obținerea cheii din textul cifrat și din cel clar. Cu toate eforturile făcute, pentru DES nu s-a găsit o metodă satisfăcătoare.

O altă cale de penetrare este să se încerce parole potențiale, pînă cînd se reușește (atac de tip „Key search"). Acest lucru este posibil deoarece oamenii au tendința de a folosi parole scurte, ușor de reținut, formate de obicei numai din litere sau date personale. Factorul critic în această metodă este creșterea timpului cerut de cifrarea parolelor potențiale și compararea lor cu intrările din fișierul de parole. De exemplu, la PDP 11/70, unde algoritmul de cifrare este astfel codificat încît să se obțină maximum de viteză, sînt necesare aproximativ 1,25 ms pentru cifrarea și verificarea unei parole. Dacă se dorește să se verifice toate parolele de lungime n, formate exclusiv din litere mici, numărul acestora este de 26^*n. Dacă se presupune că ele sînt formate din caractere tipăribile, numărul de parole posibil este de 95^*n.

Putem estima timpul de rulare a unui program care testează toate parolele, pentru un anumit număr de caractere tipăribile, așa cum s-a exemplificat în tabelul următor:

26 car. mici	36 cifre + car. mici	62 car.	95 car. tipăribile	128 car. ASCII
30 ms	40 ms	80 ms	120 ms	160 ms
800 ms	2 s	5 s	11 s	20 s
22 s	58 s	5 min	17 min	43 min
10 min	35 min	5 h	28 h	93 h
4 h	21 h	318 h		
107 h				

Analizînd tabelul, se constată că este posibil ca pe un PDP 11/70, în cîteva ore, să se obțină majoritatea parolelor cifrate de 5 sau 6 caractere, folosind doar litere mici. O

variantă a metodei anterioare de atac se bazează pe utilizarea unui dicționar de nume sau a unei liste dintr-un dicționar de cuvinte.

De exemplu, un dicționar comercial conține aproximativ 250 000 cuvinte, care pot fi testate în 5 minute. Se pot forma liste alcătuite din nume și prenume, nume de străzi, orașe, numere de telefon. Într-un studiu întreprins asupra unei mulțimi de 3 285 parole utilizate într-o lungă perioadă de timp pe un calculator PDP 11/70, s-a putut constata că:

— 15 au fost formate dintr-un singur caracter ASCII;

— 72 au fost formate din șiruri de 2 caractere ASCII;

— 464 au fost formate din șirururi de 3 caractere ASCII;

— 477 au fost formate din șiruri de 4 caractere ASCII;

— 706 au fost formate din șiruri de 5 litere, toate mari sau toate mici;

— 605 au fost formate din șiruri de 6 litere, toate mici.

În plus 492 parole apăreau în dicționare, liste de nume etc.

Un total de 2 831 sau 86% din parole intrau în una din aceste clase.

Concluzia care se desprinde de aici este aceea că utilizatorii trebuie forțați să folosească parole lungi sau alese dintr-un set mai larg de caractere. Ca urmare, programul de introducere a parolei a fost astfel modificat încît să invite la utilizarea unor parole lungi și obscure. Dacă se tastează o parolă alfabetică, mai mică de 6 caractere, sau dintr-un set mai larg de caractere dar mai scurtă de 5 caractere, programul cere introducerea unei alte parole mai lungi. Una din problemele dificile create este legată de faptul că există la ora actuală o serie de „chip"-uri comerciale care implementează DES și care fac o căutare exhaustivă a parolelor cu cîteva ordine de mărime mai rapidă. Pentru a evita o astfel de posibilitate, una din tabelele interne ale DES-ului (așa-numita tabelă E), care calculează funcția f, a fost schimbată într-un mod în care ea nu mai este constantă, ci este dependentă aleator de 12 biți care se adaugă la cheie într-o manieră ce va fi prezentată ulterior. Tabela E este setată prin apelul subrutinei „setkfy", care primește ca argument cheia (64 de biți); apelurile ulterioare

ale rutinei DES necesită 2 argumente (șirul de intrare — 64 biți — și un factor care indică funcția cifrare/descifrare) și returnează adresa șirului de 64 biți care reprezintă textul cifrat.

O altă soluție implementată în V7 pentru sporirea factorului de muncă necesar unei căutări exhaustive a parolei este următoarea: fiecare parolă este concatenată înainte de cifrare cu un număr aleator de 12 biți (numit „sare") obținut prin citirea ceasului real. Cînd se creează o parolă, P, se generează „sarea" X și se concatenează la P. În fișierul de parole se memorează atît X cît și $fXP(C)$ (11 caractere), unde f reprezintă funcția de cifrare.

Cînd un utilizator se va conecta în sistem și va introduce parola P, sistemul va extrage din fișier numărul X, va forma prin concatenare PX și va verifica $fXP(C)$ cu ceea ce este memorat în fișierul de parole. Această soluție nu determină cunoașterea factorului de muncă pentru găsirea unei anumite parole utilizate, deoarece X nu este protejat, dar sporește substanțial (de la 2 la puterea 12) deci, 4 096 ori factorul de muncă pentru generarea aleatoare de parole și verificarea lor prin compararea cu întreg fișierul de parole.

Dacă parola are n biți există 2 la puterea $n + 12$ posibilități pentru fXP, adică există 4 096 de versiuni cifrate posibile pentru fiecare parolă (*fig. 2.14*). Ca urmare, căutarea crește la cîteva zile calculator și devine impracticabil să se pregătească în avans un dicționar de parole cifrate pentru căutarea exhaustivă.

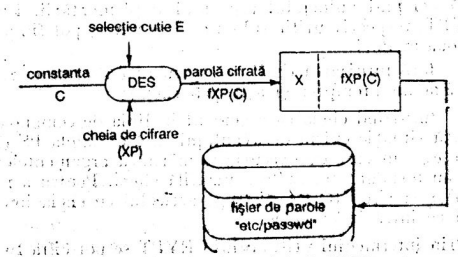


Fig. 2.14. Protecția parolelor la UNIX.

Un alt avantaj al soluției este că nu se poate detecta dacă o persoană o folosește pe mai multe calculatoare. Efortul făcut pentru protecția adecvată a parolelor este justificat deoarece acestea reprezintă unul din cele mai importante elemente care asigură protejarea mediului de execuție al fiecărui utilizator.

2.8.2.2. *Protecția fișierelor*

Pentru cifrarea datelor (fișierelor) se folosește o rutină care modelează software o mașină cu rotor de tip ENIGMA, folosită de către Germania în al doilea război mondial. Rotorul are 256 elemente diferite și permite cifrarea oricăror combinații de 8 biți. Conectarea între rotor și stator este diferită de la o cifrare la alta și depinde de cheie. Utilizatorul furnizează cheia de cifrare, pînă la 10 caractere ASCII, care este transmisă algoritmului DES, la fel ca în cazul parolelor; acesta returnează un șir de 13 caractere (din litere mari, litere mici, cifre și /) care va fi permutat și va constitui cheia mașinii ENIGMA.

Securitatea fișierelor cifrate depinde de următorii 2 factori: algoritmul de cifrare, care trebuie să fie greu de rezolvat, imposibilitatea unei căutări exchaustive a cheii în spațiul cheilor. În ceea ce privește algoritmul, deși sînt cunoscute metode de atac, factorul de muncă cerut este foarte mare. Însă alegerea cheii și securitatea cheii reprezintă cel mai vulnerabil aspect al implementării. Rutina CRYPT, respectiv utilitarul cu același nume, pot fi apelate în două moduri:

— fără furnizarea cheii ca argument (ca urmare, rutina afișează un prompter și așteaptă cheia);

— furnizînd cheia ca argument în linia de comandă; în această situație există dezavantajul că o comandă PS (lista proceselor în curs de execuție și mărimea argumentelor cu care au fost lansate) să facă vizibilă cheia. Pentru a minimiza acest risc CRYPT distruge articolul cu cheia imediat după ce intră în execuție.

Prin intermediul utilitarului CRYPT se pot cifra fișiere pentru a le face inaccesibile unor utilizatori neautorizați

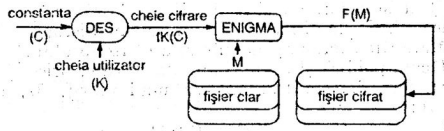


Fig. 2.15. Sistem de cifrare a datelor (fișierelor) prin comanda CRYPT.

pentru transmiterea lor sigură în rețea (fig. 2.15). În V7 editorul de texte încorporează o opțiune care-i permite să lucreze în „mod cifrare".

În acest caz, toate citirile în bufferul editorului, respectiv scrierile acestuia pe disc, sînt precedate de un prompter care cere cheia de cifrare/descifrare, datele fiind transmise/recepționate prin *pipe* ș a/de la rutina CRYPT. Folosirea fișierelor cifrate și prelucrarea lor prin utilitare specifice fișierelor text se pot face prin conectarea prin *pipe* a lui CRYPT cu aceste programe, astfel încît textul în clar să existe pe durate foarte scurte.

Pe un sistem suficient de rapid s-a obținut un *coeficient de cifrare* de aproximativ 1,47, față de transformarea nulă care s-a făcut cu aproximativ 12 000 octeți pe secundă. Din timpul de cifrare, o foarte bună parte o consumă apelarea lui DES la început și conectarea statorului la rotor printr-un generator de numere pseudoaleatoare pe baza cheii.

Acest lucru se poate observa analizînd timpii de cifrare obținuți pentru mesaje de diferite lungimi:

— 1 octet — 2,68 s;
— 3 000 octeți — 3,1 s;
— 11 000 octeți — 12,9 s.

2.8.2.3. *Poșta electronică secretă*

Poșta electronică secretă este realizată printr-un ansamblu de utilitare: XSEND, ENROLL, XGET.

***XSEND** este formată dintr-o colecție de rutine care implementează un criptosistem din categoria celor cu chei publice: **ruxac cu trapă** (metoda Merkle și Hellman). Spre deosebire de cifrurile clasice, la care se folosește aceeași

cheie atît la cifrare cît și la descifrare, la cifrurile cu chei publice, cheile de cifrare și descifrare sînt distincte. Cheia de cifrare (transformarea de cifrare a utilizatorului A), EA, este făcută publică și poate fi apelată de către orice alt utilizator care dorește transmiterea unui mesaj, M, cifrat către A:

$$C = EA(M).$$

Transformarea de descifrare (cheia de descifrare), DA, este secretă și cunoscută doar de A. Ca urmare, el este singurul în măsură să obțină mesajul în clar, M, din criptograma C:

$$M = DA(C) = DA(EA(M)).$$

Metoda Merkle și Hellman își bazează securitatea pe dificultatea rezolvării următoarei probleme: fiind dat un întreg pozitiv și un vector $A = (a_1, a_2, \ldots, a_n)$ de întregi pozitivi, să se determine un subset al lui A, a cărui sumă să fie C. Acest lucru înseamnă să se găsească un vector binar $M = (m_1, m_2, \ldots, m_n)$, astfel încît $C = A * M$ sau $C = \text{SUM}a_i * m_i$.

Vectorul A se numește „rucsac" și problema devine simplă, soluționîndu-se dacă elementele lui A sînt în strictă dominanță, adică:

$$a_i > \text{SUM}a_j.$$

Soluția unui astfel de rucsac simplu se poate face după algoritmul:

```
rucsac-simplu (A, C) este
    {
        for(i = n ; i > 1; — i)
                {
                    if c > ai mi = 1;
                    else mi = 0;
                                C = C — ai * mi
                }
              if  C = 0
                                return (M);
              else
                                return („nu există soluție");
    }
```


Merkle și Hellman au arătat cum se poate converti un rucsac simplu într-un rucsac cu o trapă, care devine greu rezolvabil fără deținerea informației trapă. Se selectează mai întîi un rucsac simplu, $A' = (a'_1, a'_2, \ldots, a'_n)$. Acesta permite obținerea facilă a soluției $C' = A' * M$. Apoi se alege un întreg, n, astfel încît

$$n > 2a_n > \text{SUM}a'_i.$$

Se alege apoi întregul w, astfel încît c.m.m.d.c. $(n, w) = 1$.

Rucsacul greu (cu trapă) se calculează astfel:

$$A = w * A' \pmod n, \text{ adică } a_i = w * a_i \pmod n.$$

Rezolvarea lui $C = A * M$ este foarte dificilă fără cunoașterea informației trapă ($w - 1$ și n), unde $w - 1$ reprezintă inversul lui w (mod n). Cu ajutorul ei se pot face transformările:

$$\begin{aligned} C' &= (w - 1) * C \pmod n \\ &= (w - 1) * A * M \pmod n \\ &= (w - 1)(w * A') * M \pmod n \\ &= A' * M \pmod n \\ &= A' * M. \end{aligned}$$

Ca urmare, cifrarea unui mesaj M se face utilizînd cheia publică (vectorul A) astfel:

$$C = EA(M) = A * M.$$

Descifrarea se face folosind cheia secretă $(w - 1, n)$ astfel:

$M = DA(C)$ — rucsac simplu $((w - 1) * C \pmod n, A')$.

Utilitarul **XSEND** va permite transmiterea unor mesaje secrete către orice alt utilizator recunoscut de sistem. Se va scoate cheia publică a destinatarului din fișier, se va cifra mesajul (cîte 6 octeți la un moment dat) și il va memora în fișierul de poștă secretă al receptorului (aici încap 10 mesaje). Apoi se va emite un mesaj la receptor: „You have mail".

*XGET este utilizată de receptorul unui mesaj secret. Acestuia i se cere o cheie (aceeași cu cea dată la ENROLL), pe baza căreia se calculează cheia secretă a receptorului. Apoi mesajul este descifrat și afișat la terminal.

* **ENROLL** este folosit de utilizatori atunci cînd doresc să se înregistreze că pot schimba poșta electronică secretă. Se cere o cheie, pe baza căreia sistemul calculează cheia publică a utilizatorului pe care o pune în fișierul:

„/usr/spool/secretmail/*.key",

unde * reprezintă numele utilizatorului.

Pe măsură ce utilizatorul va primi scrisori, se vor crea fișierele „/usr/spool/secretmail/*.0-9", de unde scrisorile pot fi descifrate prin intermediul funcției **XGET** (fig. 2.16).

În concluzie, se poate aprecia că **UNIX** este primul sistem de operare cu facilități criptografice inclusiv prin proiectare.

Ele sînt însă la un nivel de securitate mediu, scopul propus fiind mai mult să creeze pierderi considerabile celor care încearcă accese neautorizate, decît să producă invulnerabilitate.

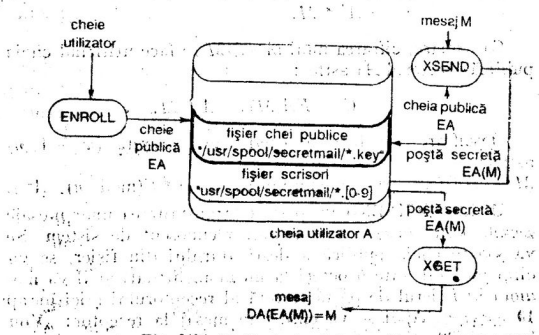


Fig. 2.16. **Poștă electronică secretă.**

Atît rutinele de cifrare a datelor, cît și de protecție a parolelor își întemeiază securitatea pe utilizarea unor parole lungi, într-un spațiu al cheilor cît mai mare. Acest lucru face ca timpul consumat pentru o căutare exhaustivă să devină prohibitiv. Algoritmii aleși, deși au slăbiciuni cunoscute de profesioniști, nu sînt triviali, și, ca urmare, sînt greu inversabili.

Pachetul de rutine criptografice de la UNIX poate fi dezvoltat în continuare pe următoarele direcții:

— implementarea unor sisteme de cifrare mai puternice și în același timp cu un coeficient de cifrare scăzut;

— implementarea altor algoritmi de cifrare cu chei publice (de exemplu, RSA) care să permită nu numai protecția poștei, ci și aplicarea semnăturii digitale;

— implementarea unui alocator de chei și parole, care să genereze la cerere parole și chei de cifrare cu lungimi și caractere corespunzătoare. Se poate implementa un astfel de generator de chei pronunțabile, ceea ce le-ar face mai ușor de reținut de către utilizatori.

2.9. NUCLEUL UNIX

Nucleul reprezintă partea rezidentă a sistemului de operare UNIX.

El asigură funcțiile de bază de supervizare și gestiune ale resurselor:

— gestiunea și planificarea proceselor;
— gestiunea și alocarea memoriei interne;
— gestiunea fișierelor;
— gestiunea perifericelor;
— tratarea cererilor (apelurilor) de funcții sistem.

Comparat cu alte sisteme de operare, nucleul UNIX furnizează un set relativ scăzut de servicii. Un utilizator însă nu face apel direct la nucleu, ci el apelează utilitare care reprezintă programe intermediare între utilizatori și nucleu. Numărul mare de utilitare și evoluția lor permanentă determină, pe de o parte, ca nucleul să fie relativ stabil, iar pe de altă parte facilitățile oferite utilizatorilor să fie foarte

bogate. Nucleul conține aproximativ 10 000 de linii sursă în limbaj C și 1 000 de linii sursă în limbaj de asamblare.

Un program de o asemenea dimensiune poate fi înțeles și întreținut de un număr mic de persoane. Acesta este și motivul răspîndirii foarte largi a UNIX-ului pe diferite tipuri de calculatoare; partea care necesită a fi rescrisă este doar cea în limbaj de asamblare. În general, nucleul se consideră ca fiind format din 2 părți importante:

— *gestiunea proceselor* — secțiune care alocă resursele, planifică procese și onorează cererile proceselor pentru funcții (apeluri) sistem;

— *gestiunea periferice lor* — care reprezintă transmisiile de date între memoria principală și periferice. Marea realizare a sistemului UNIX este că secțiunea de gestiune a proceselor funcționează identic pe orice calculator. În schimb, secțiunea de gestiune a perifericelor trebuie în bună parte rescrisă din cauza modului diferit de lucru cu perifericele al diverselor sisteme de calcul. Nucleul dispune și de o structură de date complexă, formată din diferite tabele cu care se coordonează acțiunile sale și pe care le actualizează mereu.

2.9.1. Gestiunea proceselor

Procesul reprezintă conceptul de bază în sistemul UNIX. Prin *proces* se înțelege un program în stare de execuție. Ca urmare, putem spune că procesul reprezintă unitatea de execuție. Pentru un anumit program pot exista unul sau mai multe procese active la un moment dat. În fiecare moment calculatorul execută fie un program (proces) utilizator, fie o porțiune de cod a nucleului.

Se spune despre calculator că este în *mod utilizator*, în primul caz, sau în *mod nucleu*, în cel de-al doilea. O serie de mecanisme fac comutarea calculatorului dintr-un mod în altul.

Dintre acestea amintim întreruperile de ceas, în mod uzual de aproximativ 60 de ori pe secundă. În rutina de tratare a întreruperilor de ceas se reevaluează prioritățile proceselor și este posibilă o schimbare a procesului activ; procedeul este cunoscut sub numele de „time-slicing". Cînd

un proces utilizator are nevoie de o funcție sistem, se apelează nucleul, făcîndu-se comutarea din mod utilizator în mod nucleu. În cazul operațiilor de intrare/ieșire, atunci cînd un proces lansează un astfel de schimb, el este suspendat pînă la sfîrșitul transferului. Nucleul va activa în acest timp alte procese.

2.9.1.1. *Planificarea proceselor și gestiunea memoriei*

În sistemele de operare ce lucrează în „time-sharing" procesele concură pentru obținerea unor cuante de timp în unitatea centrală.

Planificarea făcută de nucleu constă în acțiunile întreprinse de sistem pentru a decide care din procesele gata de execuție va primi dreptul de lucru în unitatea centrală în următoarea cuantă de timp. În fig. 2.17 se indică stările pe care le poate avea un proces.

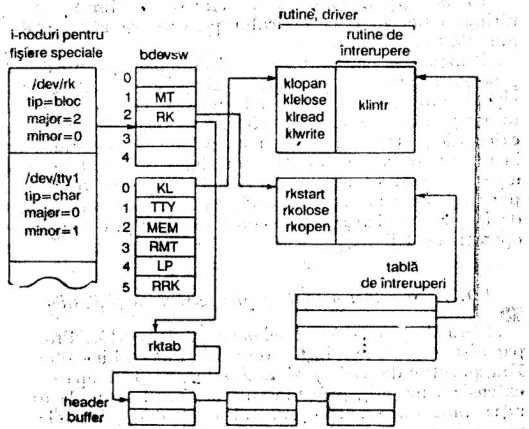


Fig. 2.17. Structurile de date ale nucleului pentru accesul la fișiere.

La un moment dat un proces este *activ* și restul ($n - 1$) sînt suspendate: *blocate* pe eveniment sau *gata* de execuție. Ideal ar fi ca toate procesele să fie simultan prezente în memoria internă.

Însă aceasta fiind o resursă limitată, nu toate procesele încap în interiorul ei. O bună parte din procesele suspendate se evacuează pe disc, iar în momentul reactivării lor prin planificare se reîncarcă în memoria internă. Acest proces de **evacuare/reîncărcare** (*swapping*) stă la baza gestiunii memoriei interne. Prin acest mecanism se eliberează spațiul de memorie ocupat de un proces prin copierea imaginii sale într-o zonă continuă pe disc (*swap-out*). Există nu anumit interval de timp cînd procesul trebuie să continue execuția, el fiind reîncărcat (*swap-in*). Algoritmul de alocare a memoriei este cu *regiune dinamică*, adică se atribuie proceselor zone de memorie de dimensiune variabilă, în funcție de dimensiunea codului, datelor și stivei procesului.

În mod uzual zona de evecuare de pe disc se află pe ultimii cilindri ai perifericului pe care se găsește sistemul de fișiere.

Pentru a se uniformiza timpul de repaus pentru diferite procese, la anumite intervale de timp se recalculează prioritățile proceselor. În acest fel procesele care au stat mult în așteptare vor avea o prioritate mai mare; la prioritatea calculată se ține seama de numărul de cuante consumate la ultima alocare a unității centrale, astfel încît procesele care au acumulat mai multe astfel de cuante vor avea o prioritate mai mică decît altele care au executat operații de I/E. Se accelerează astfel procesele lente care fac multe operații de I/E.

2.9.1.2 *Structuri de date pentru gestiunea proceselor*

Ideea de proces este fundamentală la UNIX. Procesele par niște organisme vii, fiecare avînd funcții bine stabilite. Din punctul de vedere al proceselor, nucleul apare ca o extensie a mașinii, prin funcțiile pe care le pune la dispoziția programelor. Din punctul de vedere al nucleului, procesele sînt niște structuri de date catalogate, care sînt manipulate conform unor reguli.

Lungimea în memorie a unui proces se compune din:

— *eticheta de proces*, accesibilă doar sistemului și care conține informații privitoare la starea procesului, fișiere deschise, tratarea semnalelor, precum și stiva procesului;

— *segmentele de cod* (**.text**), *date* (**.data**) și **stiva** ale programului utilizator.

Cu ajutorul editorului de legături (ld), folosind opțiunea ($-N$), se pot realiza segmente de cod reentrante, gestionate cu tabela text — TT. În cazul în care programul nu este reentrant, codul și datele sînt generate împreună în segmentele de tip **.data**, care sînt impure (neprotejate la scriere), iar segmentul **.text** $=0$.

Zona de date se compune din 2 segmente: segmentul de date inițializate de programator (**.data**) și segmentul de date neinițializate (**.bss**). Acesta din urmă poate crește dinamic prin apelarea funcției sistem **brk.** În segmentul de date (**.data**) se alocă doar datele inițializate explicit, segmentul de date neinițializat și stiva se alocă la execuție, conform informațiilor din eticheta de fișier executabil. Nucleul reține informațiile despre procese în 2 structuri de date:

— **PT** (*proces table*) — tabela proceselor;

— **UT** (*user table*) — tabela utilizator sau eticheta de proces.

Tabela PT este alocată în zona rezidentă a sistemului și conține cîte o intrare pentru fiecare proces, fiecare intrare detaliind starea procesului, identificatorul de proces (*pid*), identificatorul utilizatorului care a lansat procesul (*uid*).

Dimensiunea lui **PT** dată la generare impune numărul maxim de procese care pot coexista la un moment dat.

Tabela UT conține informații despre proces, existînd cîte o astfel de tabelă pentru fiecare proces activ. Numai **UT** a procesului activ este accesibilă rutinelor nucleului. În fig. 2.18 este prezentată legătura între aceste structuri de date.

Tabela proceselor PT este referită în momentele cruciale ale existenței unui proces. La inițierea procesului se creează o intrare în *PT* și se alocă o tabelă utilizator (*UT*) a procesului precum și segmentele de date și text ale procesului.

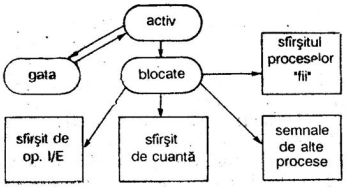


Fig. 2.18. Stările unui proces.

Ori de cîte ori un proces îşi schimbă starea (la încărcare, evacuare, în execuţie) sau primeşte semnale de la alte procese, tabela *PT* este referită. Cînd procesul se termină, intrarea în *PT* este eliberată.

Se observă că *PT* este alocată în zona rezidentă, deoarece ea trebuie să fie prezentă permanent în memorie, unele evenimente legate de proces avînd loc şi cînd acesta este suspendat şi eventual evacuat. Nucleul alocă cîte o **tabelă utilizator** (*UT*), numită şi *etichetă de proces activ*. Ea conţine informaţii care trebuie să fie accesibile atunci cînd procesul este în curs de execuţie; dacă procesul este suspendat, nu este nevoie de *UT*.

Ca urmare, *UT* face parte din regiunea alocată dinamic procesului şi este evacuată pe disc o dată cu procesul. Dintre informaţiile din *UT* amintim identificatorul utilizatorului (*uid*) şi al grupului (*gid*) pentru a determina drepturile de acces la fişier, pointer la tabela fişierelor pentru toate fişierele deschise, pointer către *i*-nodul directorului curent în tabela de *i*-noduri şi o listă de răspunsuri pentru diferite semnale.

În sistemul care dispune de dispozitiv de protecţie a memoriei, tabela *UT* nu este accesibilă utilizatorului, deşi este parte a imaginii procesului, fiind inclusă în spaţiul de adresare a nucleului. În fig. 2.18 se observă modul cum sînt gestionate procesele reentrante. Acest tip de procese are o parte de program fixă, comună pentru mai mulţi utilizatori şi o parte de date, proprie fiecărui proces. În acest fel anumite programe folosite simultan de către mai mulţi utilizatori se încarcă o singură dată, realizîndu-se o importantă economie în folosirea memoriei.

Pentru gestiunea segmentelor de program reentrante se foloseşte o structură de date numită **text table TT,** care conţine informaţiile necesare pentru segmentele de cod folosite în comun. În acelaşi timp, fiecare proces are copia zonei de date corespunzătoare programului.

2.9.1.3 *Crearea dinamică a proceselor*

Singura cale la UNIX pentru ca un proces să fie iniţiat este ca un alt proces să-l creeze. După iniţializarea sesiunii, sistemul de operare lansează un proces Shell pentru fiecare utilizator. Acesta, la rîndul său, va iniţia alte procese, creîndu-se astfel o ierarhie de procese. În fig. 2.19 procesul Shell va lansa un proces —1; acesta va iniţia procesul—2, care la rîndul său va crea procesul—3.

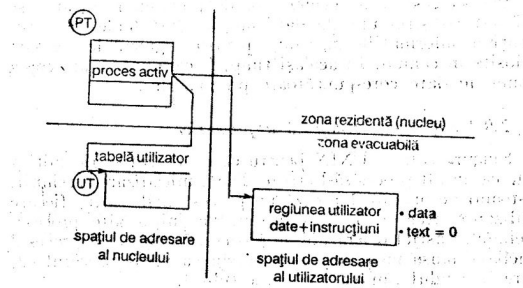
Un proces numit *tată* poate crea dinamic un nou proces, numit *fiu* prin utilizarea apelului sistem **fork.** Procesul *fiu* se obţine prin copierea în memorie a procesului *tată.* Mecanismul **fork** este singurul prin care se poate creşte numărul de procese. Dacă procesul este reentrant, procesul *tată* şi *fiu* va folosi în comun aceeaşi zonă de cod (segmentul **.text**).

Diferenţa majoră între un proces *tată* şi unul *fiu* este că au identificatori de proces diferiţi şi identificatori de procese *tată* diferiţi.

Procesul *fiu* are pid =0, iar procesul *tată* pid < >0. Ca urmare, în programul asociat procesului se poate diferenţia codul executat pentru *tată* de cel executat pentru *fiu* astfel:

```
if((p =fprk()) ==0)
        <cod executabil pentru fiu >
    else
        <cod executabil pentru tată >.
```

Cele două procese vor putea folosi în comun acelaşi fişier, deschis de procesul *tată.* Prin execuţia funcţiei **fork** se copiază întreaga imagine a procesului, inclusiv eticheta de proces (*UT*).



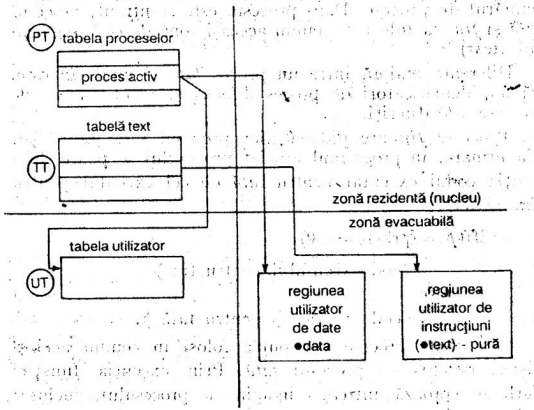


Fig. 2.19. Structuri de date pentru procese ordinare.

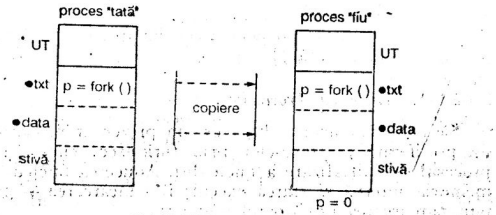


Fig. 2.20. Funcția *fork*.

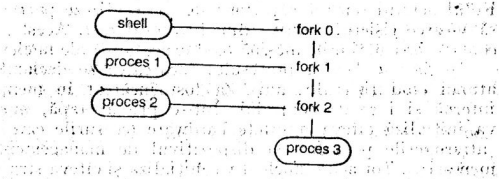


Fig. 2.21. Crearea unei ierarhii de procese.

În fig. 2.20 se prezintă o schemă de generare dinamică a unui proces. Un al doilea apel sistem foarte important în crearea proceselor este **exec**. El este folosit pentru a transforma procesul apelant într-un alt proces, prin execuția unui program cerut care înlocuiește segmentele de cod, date și stivă ale apelantului (fig. 2.21).

În urma execuției funcției **exec** procesul nu se schimbă, (are același *pid*), ci doar programul asociat acestuia este înlocuit cu un altul existent pe disc într-un fișier obiect. Prin combinarea apelurilor **fork-exec** rezultă un proces *fiu* cu o altă identitate, în care se lansează un alt program; acesta este modul de lucru tipic utilizat de Shell. Secvența de ramificare este:

```
if((p=fork()) ==0)
        {exec („fișier", ...) ...}
```

```
else
        { <cod pentru continuarea procesului
        tată >}
```

2.9.1.4 *Inițierea proceselor*

Să analizăm acum, după ce am prezentat mecanismul de proliferare a proceselor prin **fork-exec**, cum decurge procesul de inițializare a nucleului. Aducerea nucleului în memoria internă și starea execuției — **încărcarea** — presupun parcurgerea următoarelor etape:

În *prima fază* hardware-ul aduce pe disc un bloc încărcător — numit *bootstrap* — pe care-l lansează în execuție. Rolul acestui scurt încărcător este să localizeze pe disc și să încarce fișierul „/unix" din directorul *root*. Acest fișier conține instrucțiunile mașină (cod executabil) ale nucleului.

În *faza a doua* a procesului încărcarea se declanșează atunci cînd fișierul „/unix" a fost încărcat în memoria internă și i se dă controlul. Într-o primă etapă, nucleul va inițializa cîteva interfețe hardware (ceasurile care dau întreruperile periodice și dispozitivul de management al memoriei). Tot acum nucleul va inițializa și cîteva structuri de date, cum ar fi variabila care indică dimensiunea memoriei interne, bufferele bloc, bufferele *i*-nodurilor etc.

În continuare nucleul lansează **procesul** *0* (*P0*).

Toate celelalte procese se creează prin **fork.** Pentru *P0* însă mecanismul nu poate fi utilizat. Ca urmare, nucleul creează în tabela proceselor (*PT*) prima linie pe care o afectează lui *P0* și alocă o structură de date de tip *UT*, fără a avea însă cod executabil, la fel cu celelalte procese obișnuite. Procesul *P0* este creat la început și va exista pe toată durata de viață a sistemului. El se numește proces doar pentru că are afectată o intrare în *PT*, în realitate *P0* fiind doar o structură de date a nucleului.

După crearea lui *P0*, nucleul creează procesul *1* (*P1*), efectuînd o copie a lui *P0*; apoi, se extinde dimensiunea lui *P1* prin execuția codului specific apelului sistem **break.** Al treilea eveniment este crearea unui proces *P1* valabil, prin copierea în corpul său a codului specific apelului sistem **exec**, în vederea lansării în execuție a programului „/etc/

init". În acest moment inițializarea lui *F0* și *P1* este terminată: *P0* este o structură de date de tip *UT* utilizată de nucleu în procesul de planificare și gestiune a proceselor, iar *P1* este un proces valabil. În acest moment nucleul poate începe execuția *rutinelor de planificare*, procesul de inițializare fiind terminat. *Planificatorul* este responsabil cu alocarea unității centrale la procese și cu evacuarea/reîncărcarea acestora. Prima dată cînd este apelat, planificatorul ia o decizie simplă: lansează în execuție procesul *P1*, care făcînd un **exec**, acoperă propriul cod cu „/etc/init". În acest fel *P1* ajunge la forma sa finală — **proces de inițializare** (*init process*).

În continuare, procesul *P1* creează *procesul Shell* pentru a executa comenzile din fișierul „/etc/rc" de montare a fișierelor sistem, ștergere a fișierelor temporare expirate și startare a programelor de acceptare a numelor de cont de la utilizatori.

În timpul execuției UNIX-ului, *P1* va aștepta sfîrșitul tuturor *fiilor* săi.

În fig. 2.22 este prezentat ciclul de viață al procesului *P1*.

Ultimul în acțiunea de inițializare a sistemului este procesul **gtty**, care așteaptă, pentru o anumită linie, începerea servirii de către un anumit utilizator. El va lansa cu **exec** procesul de verificare a parolei, care, în caz că parola este corectă, va lansa prin **exec** programul Shell pentru acceptarea comenzilor utilizator. La sfîrșitul sesiunii, Shell va crea cu **fork-exec** un nou **proces gtty** pentru linia respectivă, proces care așteaptă începerea unei noi sesiuni.

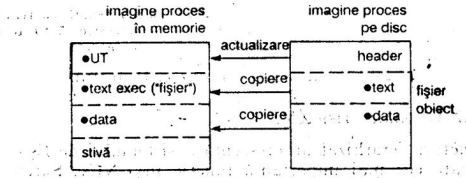


Fig. 2.22. Funcția exec.

În fig. 2.23 este ilustrat ciclul evenimentelor pentru fiecare linie.

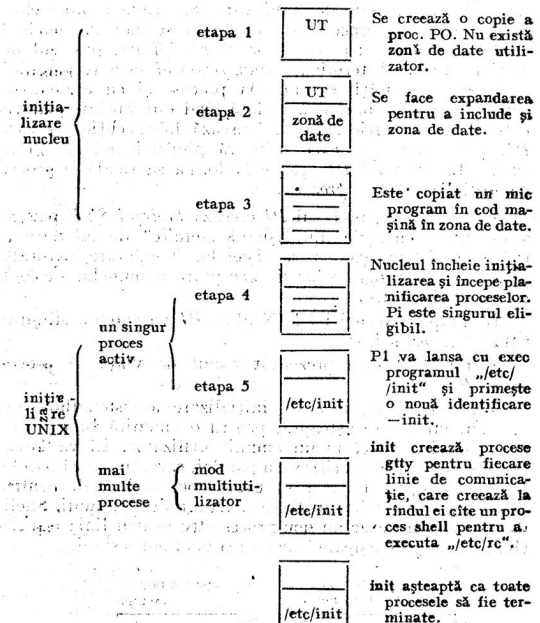


Fig. 2.23. Ciclul de viață al procesului init (P1).

2.9.2 Gestiunea fișierelor

Structura ierarhizat arborescentă a sistemului de fișiere este poate cea mai importantă funcție în UNIX. Sarcina de bază a sistemului de gestiune a fișierelor (FS—*F*ile *S*istem) este partiționarea discului și benzilor în unități numite fișiere.

Din punctul de vedere al UNIX, fiecare fișier este o succesiune de octeți. Deoarece sistemul de fișiere este ierarhic, fișierele pot fi grupate în directoare, care conțin informații despre fișiere. Directoarele sînt gestionate de către sistem.

În UNIX directoarele conțin 2 informații despre fiecare fișier: **numele** și un **număr** cu ajutorul căruia nucleul are acces la structura ascunsă a sistemului de fișiere. Structura ascunsă a sistemului de fișiere este formată din așa-numitele **noduri.**

Există cîte un *i*-nod pentru fiecare fișier și aici se găsesc atributele fișierului: localizare pe disc, lungime, mod de acces, proprietarul etc.

Oricare **volum disc** inițializat ca un sistem de fișiere (cu comanda **mkfs**) conține un număr de blocuri organizate structural pe următoarele zone (fig. 2.24):

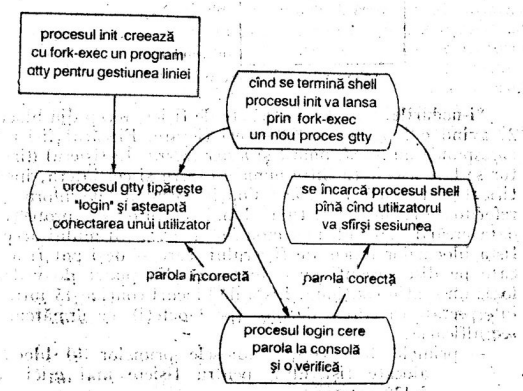


Fig. 2.24. Ciclul evenimentelor pentru o linie de comunicație.

*blocul bootstrap — conține încărcătorul care va aduce în memorie nucleul;
*superblocul — conține eticheta de volum și cuprinde:
— identificarea volumului;
— data ultimei actualizări;
— tabel cu *i*-noduri libere;
— primul element (un tabel de 50 de blocuri libere din lista spațiului liber).

Următoarele elemente din lista spațiului liber — tabele de adrese de blocuri libere — sînt conținute în blocuri libere și înlănțuite ca în fig. 2.25.

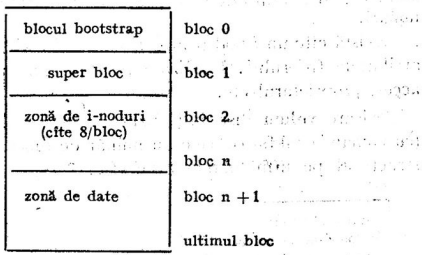


Fig. 2.25. Structura unui volum.

*i-nodurile, numite și etichete de fișier, încep din blocul 2, avînd o lungime de 64 octeți fiecare. Fiecărui fișier îi corespunde un *i-nod*, numit și *nume intern*. În fișierul director se face asocierea între numele extern și cel intern, ipentificat prin numărul de *i-nod*. Conține principalele informații referitoare la un fișier: tipul, lungimea în octeți, protecția, data creării, data ultimei modificări, data ultimului acces, lista blocurilor fizice ale fișierului, care — deși pot fi alocate pe disc necontinuu — formează din punct de vedere logic un spațiu continuu. Lista de blocuri conține 13 intrări interpretate ca adrese de bloc (pe 3 octeți), cu următoarea semnificație:

— primele 10 — conțin adresele primelor 10 blocuri alocate fișierului, pentru fișiere mai mici de 5 120 octeți;

— intrarea 11 — conține adresa blocului de simplă indirectare, care conține adresa a 128 de blocuri de date (pentru fișiere pînă la lungimea de 70 656 octeți);
— intrarea 12 — conține adresa blocului de dublă indirectare, care conține adresele a 128 de blocuri date de (pentru fișiere pînă la 8 459 264 octeți);
— intrarea 13 — conține adresa blocului de triplă indirectare, care conține adresa a 128 blocuri de dublă indirectare (pentru fișiere de lungime maximă de 1 082 201 087 octeți).

Acest mecanism de adresare, prezentat în fig. 2.26 permite extragerea unui bloc de date într-un număr de 1—4 accese. După cum se poate observa, sînt favorizate fișierele mici, la care accesul are un număr mai mic de indirectări.

La fișierele speciale, *i*-nodul conține atribute ale fișierului în primele două intrări, restul rămînînd nefolosite.

În fig. 2.27 este ilustrat modul în care sistemul parcurge o anumită cale pînă la localizarea unui fișier. Am analizat drept exemplu calea „../a/b" care pleacă din directorul curent la directorul *tată* (..), apoi la subdirectorul acestuia (*a*) și apoi la fișierul căutat (*b*). Pentru a urma această cale, sistemul de operare parcurge următorii 8 pași:

1. se încarcă *i*-nodul directorului curent (adresa sa este în UT);
2. se caută în directorul curent intrarea „.." scoțînd de aici numărul *i*-nodului director *tată*;

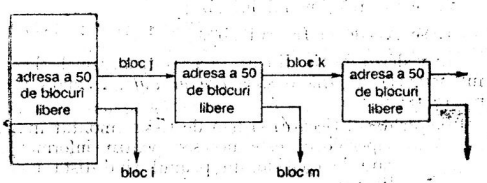


Fig. 2.26. Înlănțuirea blocurilor libere.

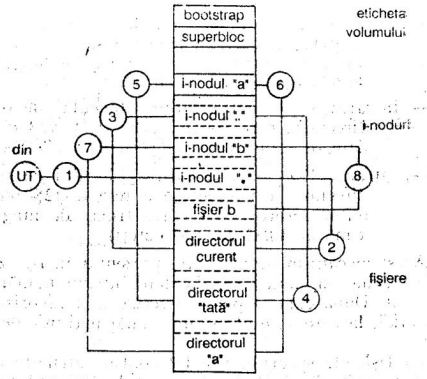


Fig. 2.27. Înlănţuirea blocurilor alocate unui fişier (blocurile sînt numerotate în secvenţă logică: fizic, ele sînt împrăştiate pe disc).

3. se încarcă *i*-nodul directorului *tată*;

4. se caută aici intrarea „a", extrăgîndu-se numărul *i*-nodului directorului „a";

5. se încarcă *i*-nodul lui „a";

6. se caută în directorul „a" fişierul „b", extrăgîndu-se numărul *i*-nodului „*b*";

7. se încarcă *i*-nodul lui „b";

8. se accede la fişierul „b".

În vederea gestiunii corecte a fişierelor, nucleul menţine în memoria internă o *structură de date*, formată din mai multe tabele:

***superblocul** fiecărui sistem de fişiere montat în arbore; superblocul este necesar pentru informaţiile pe care le conţine, în primul rînd lista blocurilor libere;

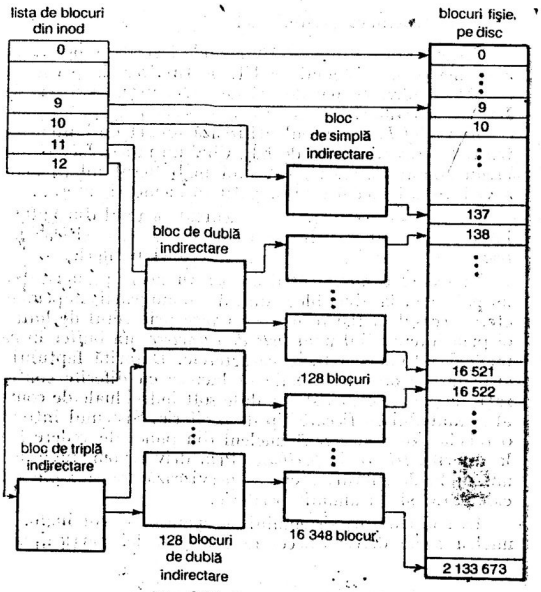


Fig. 2.28. Parcurgerea unei căi.

***tabela fişierelor** (*FT*), care conţine cîte o intrare pentru fiecare fişier deschis, intrare care pointează structura de date de tip *i*-nod a fişierului respectiv. De asemenea, în *FT* se află un pointer de citire/scriere;

***tabela *i*-nodurilor** (*IT*), ce conţine *i*-nodurile necesare pentru accesarea fişierelor deschise de procese. Ţinerea lor în memoria internă permite o localizare mai rapidă a blocurilor alocate unui anumit fişier.

În fig. 2.28 se reprezintă aceste tabele ale nucleului.

2.9.3 Gestiunea perifericelor

Sistemul de operare UNIX include 2 strategii de gestiune a perifericelor: sistemul de I/E de tip *bloc*, la perifericele ce pot fi adresate pe blocuri de 512 octeți, și sistemul de I/E de tip *caracter*.

În *modul bloc*, nucleul utilizează un set de buffere pentru reducerea traficului de I/E. Cînd un proces face o cerere la un anumit periferic, se caută în bufferul intern pentru a vedea dacă blocul este pregătit în memoria internă.

Dacă blocul cerut nu este încărcat în unul din bufferele interne, nucleul va elibera unul din buffere și va iniția transferul cerut între bufferul intern și periferic.

În *modul caracter*, utilizat pentru acele periferice care nu pot lucra la nivel bloc (linii de comunicații, imprimante etc.), accesul la fișiere nu se va face prin setul de buffere, ci prin intermediul unei *liste de caractere*, un buffer în care nucleul introduce/extrage caracterele. Datorită faptului că sistemul de operare trebuie să lucreze cu diferite configurații de periferice, există module soft individuale de control al schimbului cu fiecare tip de periferic. Sistemul întreține o tabelă care conectează nucleul din punct de vedere logic la diferite *drivere de periferic*. Prin driver vom înțelege un ansamblu de subrutine care supervizează transmisiile între calculator și un anumit periferic.

La configurarea sistemului de operare, se vor îngloba în nucleu acele drivere necesare configurației particulare în

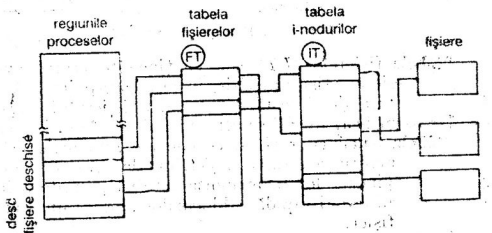


Fig. 2.29. **Structura de date a nucleului pentru accesul la fișiere.**

care lucrează sistemul. Accesul la un anumit driver este dat de *numărul major* care corespunde tipului de periferic. Pentru definirea univocă a perifericului este necesar și *numărul minor*, care indică numărul de unitate al dispozitivului.

Sistemul de operare folosește 2 tabele: **bdevsw** și **cdevsw** pentru a controla operațiile de I/E (fig. 2.29):

- ***cdevsw** — este o structură de date care face legătura între modulele driverelor pentru perifericele mod caracter și nucleu, conținînd adresa driver pentru fiecare număr de periferice;
- ***bdevsw** — realizează același lucru pentru perifericele în mod bloc. În plus, **bdevsw** conține și adresa unei tabele de periferice (de exemplu, **rktab**), care conține pointeri pentru bufferele asociate perifericului.

Capitolul 3

SISTEMUL DE OPERARE VMS

Sistemul de operare VMS funcţionează pe minicalculatoarele VAX de 32 biţi, extensie a unuia din cele mai populare minicalculatoare, PDP-11, avînd cuvîntul de 16 biţi. Ambele sînt realizări ale firmei DEC.

3.1. INTRODUCERE ÎN VAX ŞI VMS

VAX reprezintă acronimul pentru *V*irtual *A*ddress e*X*tension. Termenul de „adresă virtuală" sugerează că VAX poate folosi adrese de memorie (locaţii) care nu există în memoria fizică. Acesta reprezintă principalul scop al arhitecturii VAX-11, şi anume creşterea spaţiului virtual disponibil pe PDP-11.

O altă caracteristică a arhitecturii VAX este aceea că permite multiprogramarea. Diferiţii utilizatori, lansînd în execuţie diferite programe, pot folosi simultan sistemul, fiecare avînd senzaţia controlului deplin asupra resurselor. În realitate VAX va prelucra porţiuni din fiecare program utilizator, cîte unul la un moment dat.

VAX reprezintă o familie de calculatoare, diverse ca proiectare, facilităţi şi tehnologie. Însă sistemul de operare VAX/VMS se poate executa pe orice membru al acestei familii: minicalculatoarele VAX-11/700 (780, 750, 730), superminicalculatoarele din seria VAX 8000 şi supermicrocalculatoarele MICROVAX.

VAX-11/700 reprezintă descendentul lui PDP-11, fiind proiectat astfel încît să se asigure totala compatibilitate cu PDP-11, sub aspect periferic şi software, însă cu o altă tehnologie şi cu o viteză mult sporită.

VAX-8000 reprezintă o nouă generaţie de minicalculatoare DEC cu o nouă arhitectură a intrărilor/ieşirilor, ceea ce asigură performanţe deosebit de ridicate şi configuraţii flexibile. Ele se bazează pe arhitecturi paralele de tip asimetric cu un mare număr de procesoare „master-slave".

Calculatoarele MICROVAX furnizează aceleaşi facilităţi hardware şi software ca şi VAX, dar într-o configuraţie mai mică şi mai ieftină. Ele sînt destinate să fie utilizate ca microcalculatoare personale.

Din punctul de vedere al utilizatorilor, singura diferenţă semnificativă între calculatoarele familiei VAX este viteza de prelucrare, în rest orice program realizat pe unul din calculatoare se poate executa pe oricare alt membru al familiei.

Există multe elemente de compatibilitate între sistemele VAX şi PDP-11. În primul rînd sintaxa şi mnemonicele limbajului de asamblare sînt în cea mai mare parte similare. Compilatoarele care generează cod PDP-11 pot fi uşor adaptate, astfel încît să genereze cod VAX-11. Sistemul de operare VMS este similar în multe privinţe cu RSX. Organizarea sistemului de fişiere este identică. Există un pachet de subrutine, numit Applications Migration Executive, care ajută utilizatorii în execuţia pe VAX a unor programe PDP-11, prin emularea serviciilor RSX.

Familia VAX reprezintă în momentul de faţă liderul industriei de minicalculatoare. În multe alte ţări s-au proiectat şi implementat arhitecturi compatibile VAX. Şi în ţara noastră s-a realizat minicalculatorul CORAL 8730, compatibil cu VAX 11/730.

Dintre noile facilităţi VAX putem aminti arhitecturile multimini care pot cuprinde pînă la 16 procesoare VAX ce lucrează împreună, utilizarea în reţea DECNET a minicalculatoarelor VAX, realizarea unor staţii de lucru VAX (calculatoare personale) şi a unor superminicalculatoare răcite cu lichid, de mare performanţă.

Una din facilitățile cele mai importante la arhitecturile VAX este aceea de CLUSTER, adică, un ansamblu de mai multe calculatoare VAX puternic interconectate printr-un cuplor de mare viteză. Astfel de configurații duc la obținerea unor puteri de calcul mari și la o foarte bună partajare a resurselor.

SISTEMUL DE OPERARE al familie VAX este VMS (*V*irtual *M*emory *S*ystem) — un sistem de operare *INTERACTIV*, care permite lucrul în multiprogramare al mai multor utilizatori simultan. Modul în care se execută *MULTIPROGRAMAREA* este puțin diferit decît în abordările tradiționale. În aceste cazuri se alocă cuante de timp UC fiecărui utilizator din sistem, folosindu-se un algoritm de rotație care permite satisfacerea egală a tuturor programelor. În cazul lui VMS, fiecare utilizator primește o cuantă de timp UC, în funcție de necesitățile sale și de prioritatea sa. Dacă un utilizator nu folosește cuanta alocată sau o porțiune din ea, aceasta este alocată următorului utilizator. Acest mecanism de planificare este foarte eficient.

O caracteristică fundamentală a sistemului VMS este implementarea conceptului de MEMORIE VIRTUALĂ. Acest lucru permite executarea unor programe mari care pot adresa pînă la 4 octeți. Dar aceasta este memoria virtuală, memoria fizică putînd avea limite mai mici, ea trebuind să fie împărțită și între mai multe programe. Sistemul de operare, prin mecanismele de paginare și evacuare/încărcare, va aduce în memoria fizică acele porțiuni de programe necesare execuției imediate. Cealaltă parte a programelor, care nu este necesară în acel moment, se va afla rezidentă pe disc.

Deoarece se pot executa simultan mai multe programe, fiecare va avea propriul său spațiu virtual. Acesta este împărțit în pagini, de 512 octeți fiecare; un mecanism bazat pe o tabelă de pagini, face corespondența între fiecare pagină virtuală (adresa virtuală) și adresa fizică corespunzătoare. Sistemul VMS furnizează 4 NIVELURI DE PROTECȚIE (fig. 3.1): nucleu, executiv, supervizor, utilizator.

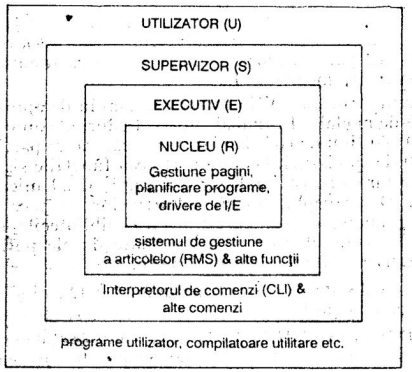


Fig. 3.1. Niveluri de protecție în VMS.

Asupra fiecărei locații de memorie, cele 4 categorii de programe pot avea acces de scriere/citire, citire sau nu pot avea acces.

Utilizatorii interacționează direct în sistemul de operare prin intermediul unui *LIMBAJ DE COMANDĂ* DCL (*D*igital *C*ommand *L*anguage). Acesta este preluat direct de către *INTERPRETORUL DE COMENZI* CLI (*C*ommand *L*anguage *I*nterpreter). Noțiunea fundamentală a sistemului VMS este aceea de *PROCES*, care reprezintă contextul hardware și software al unui utilizator conectat în sistem. Procesul asociat utilizatorului este creat la intrarea în sesiune și are o serie de atribute ca:

— numele procesului;
— identificatorul;
— codul de identificare al utilizatorului (UIC);
— privilegiile etc.

Toate acțiunile inițiate în timpul sesiunii de lucru generează încărcarea unor *PROGRAME* sau *IMAGINI*. O

imagine este un ansamblu de instrucțiuni mașină, care rezidă într-un fișier creat de către link-editor. Încărcarea imaginii presupune și alocarea unor resurse ca drepturi de acces, procesor, memorie, I/E.

Sistemul de operare VMS este un ansamblu de aproximativ 8 000 de module. În limbaj sursă, el cuprinde aproximativ 5 500 000 linii de cod, la elaborarea cărora au participat peste 200 de specialiști. O mare parte din funcțiile sistemului de operare au fost implementate hard, fiind microprogramate. De exemplu, apelul subrutinelor se face prin microprogramare, ceea ce a permis ca bibliotecile să fie unice în sistem, subprogramele conținute de ele putînd fi apelate din orice limbaj de programare.

Istoria VMS-ului datează de aproximativ 12 ani și poate fi rezumată astfel:

— 1977: apare *VMS-V1*, în care sînt înglobate concepte de la cele mai răspîndite sisteme de operare, ca UNIX, RSX, CP/M. Sistemul era compatibil cu PDP, iar singurul limbaj disponibil era FORTRAN;

— 1980: apare *VMS-V2* — care se desprinde de RSX și devine sistem de operare autonom. S-a introdus un limbaj de comandă (DCL) și un foarte bun compilator de COBOL;

— 1982: apare *VMS-V3* care reprezintă o versiune de tranziție spre cea actuală;

— 1983: o dată cu MICROVAX apare și sistemul *MICRO VMS* compatibil cu V3;

— 1984: apare una din cele mai importante versiuni *VMS-V4*, în care este înglobat conceptul de *cluster*, adică de gestionare a unor configurații stea de multiminicalculatoare (max 16), cuplate în jurul cuplorului stea (fig. 3.2).

În cadrul unor astfel de arhitecturi se poate folosi un singur disc sistem, comun tuturor calculatoarelor, interconectate printr-un controler inteligent (HSC-50) de mare viteză (70 Mbiți/s);

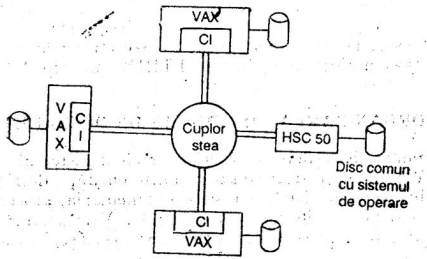


Fig. 3.2. Arhitectură de tip cluster.

— iunie 1988: apare *VMS-V5*, versiune regîndită a sistemului de operare, cu un grad ridicat de paralelism oferit de cluster sau de rețea DECnet. De asemenea, se aduc îmbunătățiri semnificative în gestiunea memoriei virtuale.

În cadrul lui V5, DECnet este integrat total în sistemul de operare; ca urmare, operațiile de intrare/ieșire sînt transparente la nivel utilizator, acesta procedînd similar și cu perifericele locale, precum și cu cele la distanță. Configurația rețelei este prezentată în fig. 3.3.

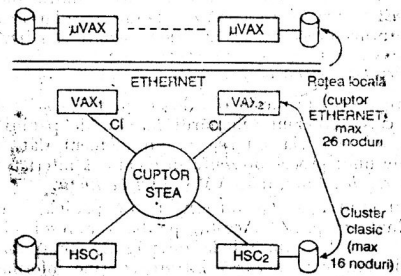


Fig. 3.3. Integrare DECnet în VMS.

Firma DEC a implementat în V5 un compilator FORTRAN VS care generează cod specific calculatoarelor multiprocesor. De asemenea, se investește mult pentru dezvoltarea sistemului de operare ULTRIX (de tip UNIX)

3.2. ORGANIZAREA SISTEMULUI DE OPERARE VMS

Din punct de vedere structural, nucleul sistemului de operare VMS este alcătuit din mai multe entități, destinate controlului și administrării resurselor: memoria, procesele, intrările/ieșirile (perifericele) și fișierele. Vom analiza pe scurt modul în care sînt rezolvate aceste cerințe în cadrul VMS.

3.2.1 Gestiunea memoriei

În gestiunea memoriei interne se utilizează conceptul de memorie virtuală. Există două module, numite *pager* și *swapper*, care implementează aceste funcțiuni în VMS. *Pager* este însărcinat cu citirea unei pagini, atunci cînd apare o întrerupere generată de lipsa paginii în memoria internă sau cu scrierea unei pagini modificate din memoria internă pe disc, atunci cînd se dorește crearea unui spațiu pentru încărcarea unei pagini noi. *Pager* este o subrutină folosită în comun de toate procesele din sistem.

Swapper încarcă sau evacuează procese în/din memoria internă. El lucrează în strînsă colaborare cu planificatorul, care determină procesele ce trebuie evacuate sau încărcate.

**Încărcarea/evacuarea paginilor*

Fiecare proces primește o valoare numită *limita setului rezident*, care reprezintă numărul maxim de pagini ce se pot afla în memoria internă la un moment dat. Aceste pagini ale unui proces prezente în memoria internă alcătuiesc *setul rezident*, numit în VMS și *set de lucru*.

Atunci cînd apare o întrerupere de pagină, controlul este preluat de pager. Acesta, pe baza adresei virtuale, determină intrarea din tabela de pagini, corespunzătoare paginii de încărcat. Această intrare conține informații cu privire la amplasarea paginii pe disc.

Apoi *pager* localizează o pagină liberă în memorie și lansează procesul de citire a paginii de pe disc. Intrarea din tabela de pagini este modificată astfel încît să indice adresa paginii în memoria internă. Apoi *pager* redă controlul la instrucțiunea care a generat întreruperea.

În timpul execuției unui proces, se generează astfel de întreruperi de fiecare dată cînd se referă o pagină care nu este în setul rezident. Dacă valoarea limită a setului rezident este activă, procesul va trebui să evacueze una din paginile sale pentru încărcarea paginii apelate.

Este interesant de observat că VMS nu folosește un algoritm sofisticat de selectare a paginii ce urmează a fi înlocuite; se folosește în acest scop un simplu *algoritm de tip FIFO*. Acest lucru poate crea o anumită întîrziere în execuția unor programe.

Sistemul de operare ține la zi două liste: *lista paginilor libere* și *lista paginilor modificate*. Cînd un proces are nevoie de o nouă pagină liberă, se caută în lista paginilor libere. Desigur că toate paginile memoriei interne pot fi ocupate. Cînd un proces trebuie să elimine o pagină proprie, deoarece a atins limita setului rezident, aceasta este pusă fie în lista paginilor libere (dacă nu a fost modificată prin scriere), fie în lista paginilor modificate. Paginile modificate nu sînt scrise pe disc pînă cînd lista corespunzătoare nu ajunge la o anumită dimensiune. Orice pagină care se află în lista paginilor modificate poate fi cerută din nou de proces, ceea ce reduce într-o oarecare măsură traficul de pagini schimbate între memorie și disc.

Trebuie subliniat faptul că VMS actualizează în mod dinamic lista setului rezident, în funcție de numărul de întreruperi de pagini care apare pentru un anumit proces. În momentul cînd se constată un număr mare de întreruperi, se mărește valoarea limitei; dacă numărul scade, limita este redusă. În acest fel se tinde să se egalizeze rata întreruperilor de pagină pentru toate procesele.

**Încărcarea/evacuarea proceselor*

În afara procesului de evacuare/încărcare de pagini, sistemul de operare realizează prin *swapper*, evacuarea/încărcarea întregului set rezident de pagini ale unui proces

pe/de pe disc. Evacuarea setului se face atunci cînd se dorește să se creeze spațiu pentru setul unui alt proces. Spre deosebire de alte sisteme de operare, VMS încarcă întreg setul rezident al unui proces, set pe care acesta îl avea în momentul evacuării.

VMS ia măsurile necesare, astfel încît procesul încărcat să primească cel puțin o cuantă de timp în UC înainte de o eventuală evacuare a sa.

Swapper, atunci cînd dorește să lanseze încărcarea unui proces în memoria internă, va trebui să găsească un număr suficient de pagini libere pentru a primi întregul set rezident al procesului. Paginile libere sînt determinate în mai multe moduri: din lista de pagini libere, din lista de pagini modificate, prin scrierea acestora pe disc sau prin evacuarea întregului set rezident al unui proces cu prioritate mai mică (sau egală) decît prioritatea procesului încărcat. Uneori este necesară evacuarea mai multor procese pentru a se crea spațiul de încărcare a unui proces cu un număr mai mare de pagini rezidente.

3.2.2 Planificarea proceselor

VMS utilizează o schemă de planificare a proceselor executabile bazate pe priorități. Există procese normale, numite și *procese multiprogramate*, dar și procese în timp real, numite *procese în timp critic*.

VMS utilizează 32 *de cuvinte de prioritate* pentru planificare (fig. 3.4). Prioritățile 0—15 se alocă proceselor normale, iar prioritățile 16—31, proceselor în timp real.

Numerele mari indică priorități ridicate. Procesele în timp real sînt planificate strict după priorități. Prioritățile proceselor normale sînt însă modificate dinamic, astfel încît să se atingă o utilizare eficientă a UC. Fiecare eveniment sistem are asociat un *eveniment de prioritate*.

Atunci cînd apare evenimentul care face ca un proces să devină executabil, planificatorul setează prioritatea procesului la valoarea de bază, plus incrementul asociat acelui eveniment.

De fiecare dată cînd un proces este planificat în vederea execuției unui proces, prioritatea sa descrește, dar niciodată sub valoarea sa de bază (fig. 3.5).

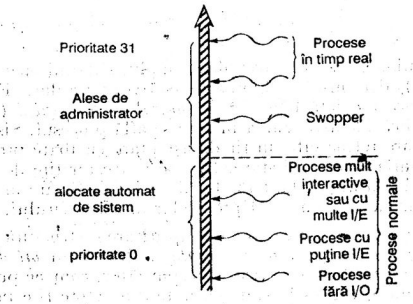


Fig. 3.4. Prioritatea proceselor

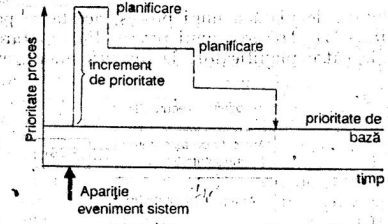


Fig. 3.5. Ajustarea dinamică a priorităților proceselor normale.

Evacuarea/reîncărcarea proceselor din/în memoria internă este necesară din mai multe motive: să înlocuiască procesele cu prioritate mică cu altele cu prioritate mai mare; să înlocuiască procese care așteaptă evenimente cu altele de pe disc care pot fi executate.

VMS asigură un timp de răspuns cît mai scurt proceselor în timp real, care au prioritate mai mare decît cele normale. Următorul proces ales spre execuție este cel cu priori-

tatea cea mai mare în firul de aşteptare al proceselor rezidente executabile.

Evenimentele care apar (de exemplu, sfîrşitul operaţiilor de I/E) determină schimbarea stării proceselor. Fiecare proces are asociată o *tabelă de control al procesului* (PCB), care indică inclusiv starea în care se află procesul. Sistemul de operare reţine cîte un fir de aşteptare cu toate procesele care se află în aceeaşi stare. În cadrul acestor fire de aşteptare se înlănţuie de fapt PCB-urile. Fiecare fir are un antet care conţine un contor al proceselor asociate firului.

În fig. 3.6 se indică spre exemplificare o structură tipică a *firelor asociate proceselor executate*. Există un *bit de prezenţă* care, dacă este pe firul corespunzător, conţine procesul în aşteptare. Procesele sînt selectate din aceste fire în ordinea priorităţii. În cadrul unui fir de aşteptare, unde toate procesele au aceeaşi prioritate, selecţia se face după un algoritm FIFO.

Pe durata de viaţă a unui proces, acesta se poate afla în diferite stări. Trecerea unui proces dintr-o stare în alta se face de către planificator, la apariţia unor evenimente.

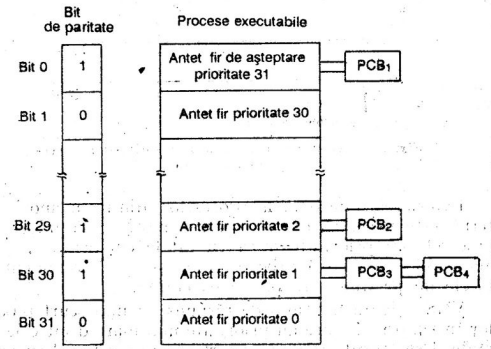


Fig. 3.6. Firele de aşteptare pentru procesele executabile.

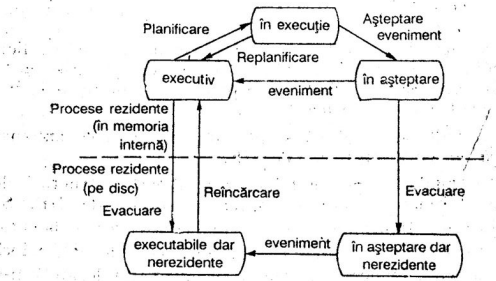


Fig. 3.7. Diagrama de stări a proceselor.

În fig. 3.7 se prezintă diagrama de stări corespunzătoare proceselor în VMS.

Un proces în curs de execuţie poate fi replanificat în stare executabilă dacă a consumat cuanta alocată. O serie de operaţii cer ca procesul curent să intre în stare de aşteptare, pînă la producerea evenimentului aşteptat. Un proces nou creat va intra în starea executabil nerezident.

Cuanta de timp alocată de către sistem fiecărui proces are două roluri: indică un timp minim de prezenţă a procesului în memorie înainte de a fi evacuat şi, în acelaşi timp, forţează o utilizare egală a UC de către toate procesele. Procesele în timp real nu sînt însă controlate cu privire la expirarea timpului. De notat că la VMS noţiunea de *cuantă* defineşte timpul de ocupare a memoriei interne şi numai indirect utilizarea UC.

3.2.3 Gestiunea intrărilor/ieşirilor

În cadrul VMS, sistemul intrare/ieşire este organizat pe mai multe niveluri. Operaţiile coboară de la nivelul logic la cel fizic. La cel mai înalt nivel VMS prelucrează fişiere, articole şi cîmpuri. La nivelul de jos el este însărcinat cu operaţiile fizice şi organizarea diferitelor tipuri de periferice.

În general, operațiile de I/E au un caracter asincron: odată lansată o operație de I/E, procesul poate continua la execuție. Procesul trebuie însă să verifice în mod explicit terminarea operației de I/E. Dacă se dorește așteptarea unei I/E procesul trebuie să se pună în așteptare, utilizînd în acest scop o rutină Sistem.

Trebuie subliniat că suspendarea nu se face **automat** după lansarea operației de I/E.

Sistemul de operare folosește mai multe structuri de date destinate și descrie starea sistemului de I/E. Se utilizează blocuri de control care descriu fiecare adaptor de bus, controlor, unitate fizică și toate cererile de I/E încă nerezolvate. Diferitele blocuri de control sînt legate într-o structură care reprezintă topologia sistemului hardware de I/E.

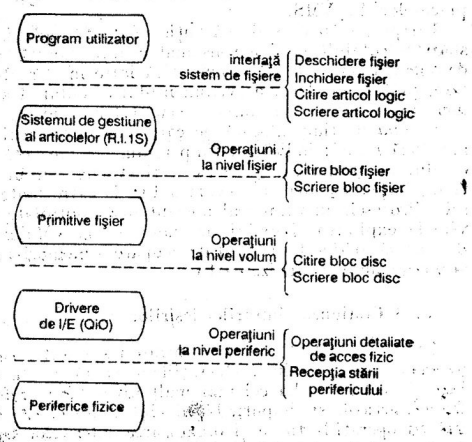


Fig. 3.8. Niveluri în sistemul de intrare/ieșire.

Principalele blocuri de control sînt următoarele:

***DDB** (*D*evices *D*ata *B*lock) — conține informații comune la mai multe periferice de același tip care sînt conectate la un singur controlor (ca de exemplu, numele perifericului etc.);

***IRP** (*I*/*O* *R*equest *P*acket) — conține toate informațiile unei cereri de descriere de I/E (ca de exemplu, identificatorul procesului apelant, operația cerută, adresa și lungimea bufferului utilizator);

***UCB** (*U*nit *C*ontrol *B*lock) — conține caracteristicile și starea unui anumit periferic și contextul procesului driver pentru acel echipament;

***CRB** (*C*hannel *R*equest *B*lock) — conține informații care descriu starea controlorului. Se reține identificatorul perifericului care transferă date în acel moment și identificatorii perifericelor care așteaptă eliberarea controlorului;

***IDB** (*I*nterrupt *D*ispatch *B*lock) — indică ce face la un moment dat un controlor, fiind folosit pentru determinarea perifericului care a dat o anumită întrerupere;

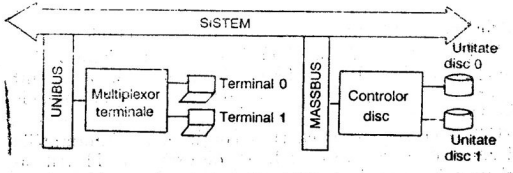
***ACB** (*A*daptor *C*ontrol *B*lock) — conține informații despre starea Unibus-ului sau a Massbus-ului.

În figura 3.9*a* este prezentat un exemplu de configurație hardware, iar în 3.9*b* se indică structura de date care reprezintă starea acestei configurații.

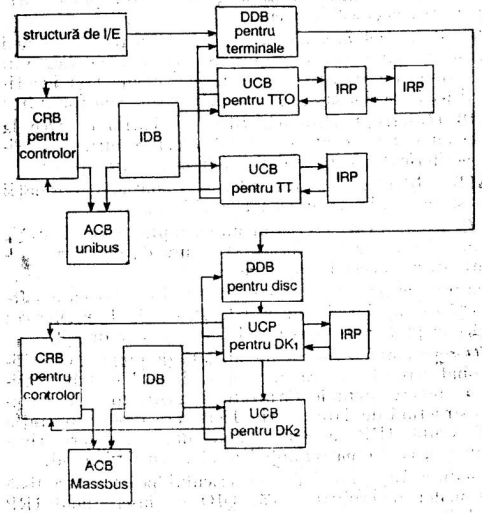
O intrare/ieșire este prelucrată de trei componente software: QIO — apelul sistem de I/E; driverele de periferice; rutinele de prelucrare a sfîrșitului operațiilor de I/E.

QIO reprezintă o procedură apelată de programul utilizator cînd cere efectuarea unei operații de I/E. QIO validează toate argumentele furnizate de utilizator și construiește o structură de date de tip IRP care descrie operația de I/E cerută. IRP va conține toate datele necesare driverului pentru a efectua schimbul de date cu perifericul.

Structura UCB asociată perifericului pointează o listă a IRP-urilor nerezolvate încă. QIO va insera noul IRP în firul de așteptare corespunzător perifericului la care este făcută cererea, iar apoi redă controlul procesului apelant. Aceasta poate continua sau poate executa o cerere de suspendare pînă la sfîrșitul operației de I/E.



a



b

Fig. 3.9. a — configurația hardware; b — structura de date de I/E.

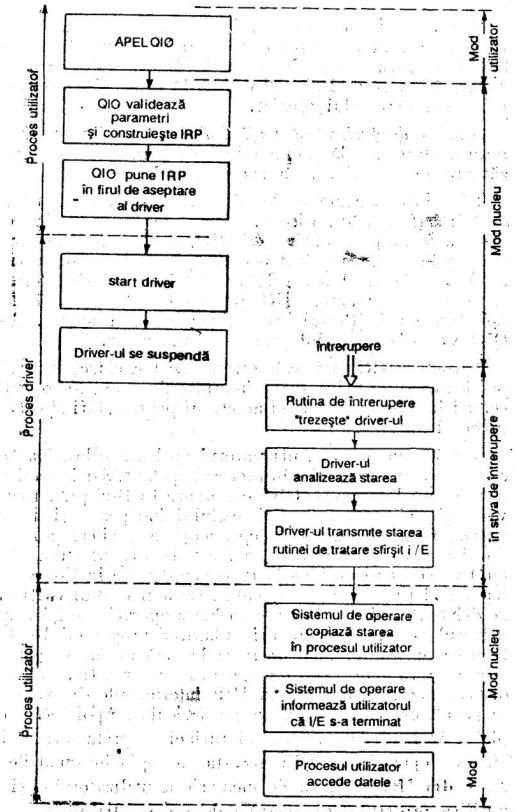
Driverele controlează funcționarea diferitelor periferice. Ele sînt executate cu prioritate ridicată pentru a accelera lucrul cu perifericele. Structura UCB definește un proces de tip driver în același mod în care PCB descrie starea acestuia și a driverului asociat.

Rutina de prelucrare a sfîrșitului operațiilor de I/E va returna starea finală și datele în zona de memorie corespunzătoare procesului care a executat I/E.

Să analizăm acum în detaliu care este *traseul unui I/E în VMS* (fig. 3.10). Fiecare cerere de intrare/ieșire este inițiată de un proces prin apel QIO, scris explicit în limbaj de asamblare sau rezultat prin compilarea unor instrucțiuni în limbaj de nivel înalt. Procesul QIO va valida cererea, va construi o structură IRP pe care o va pune în firul de așteptare al UCB-ului asociat perifericului apelat. Apoi QIO va reda controlul procesului utilizator.

Structura IRP definește cererea de I/E către sistem. În particular ea conține toate informațiile necesare execuției cererii în mod asincron și accesului pentru stări și date în procesul apelant.

După cum se vede, utilizatorul apelează subrutina QIO care se va executa chiar în procesul apelant, dar în mod nucleu (ca mod de acces). QIO localizează UCB-ul perifericului vizat și verifică parametrii apelului independenți de periferic. Pentru parametrii dependenți de periferic apelează diferite subrutine driver care fac validarea acestora. Apoi QIO va construi un bloc IRP pe care-l pune în firul de așteptare al UCB-ului perifericului. De aici IPB-ul va fi luat de driverul perifericului care va lansa operația de I/E. Apoi driverul se va suspenda prin plasarea propriului UCB în firul de așteptare. Cînd operația lansată de driver s-a terminat, se va genera o întrerupere care va reactiva procesul driver. Aceasta va copia în IRP informațiile de stare din registrele controlorului și ale perifericului. Apoi IRP va fi plasat în firul de așteptare al rutinei de prelucrare a sfîrșitului operației de I/E. Aceasta va copia informațiile de stare din IRP în zona de memorie a utilizatorului și va notifica procesului că operația de I/E s-a terminat.



Figa 3.10. Etapele de prelucrare a unei cereri de I/E

3.2.4. Sincronizarea și cooperarea proceselor

VMS furnizează o serie de facilități de comunicare între procese în vederea sincronizării acestora, transmiterii de mesaje între ele sau cooperării prin zona de date comune. Se pot folosi patru tehnici diferite de comunicație:

— prin fanioane eveniment;
— prin cutii de scrisori;
— prin zone comune de memorie;
— prin fișiere utilizate în comun.

Fanioane eveniment. Fanioanele eveniment reprezintă cel mai simplu mecanism de comunicare interproces în VMS. Un fanion eveniment este un bit care poate fi setat sau șters de către proces. Acesta are la dispoziție următoarele facilități legate de fanioane:

— setarea sau ștergerea unui fanion eveniment;
— tastarea stării unui fanion eveniment;
— plasarea programului într-o stare de așteptare pînă cînd un anumit fanion sau un grup de fanioane vor fi setate.

Fanioanele eveniment rezidă în registre de 32 de biți. Fiecare proces are acces la 4 registre de fanioane, numerotate de la 0 la 3. Două dintre ele sînt destinate unor funcții locale, iar două, numite comune, comunicării între procese.

Registrele 2 și 3 conțin fanioane create dinamic prin cereri sistem, care sînt folosite în comun cu alte procese. VMS pune la dispoziția proceselor 6 servicii pentru accesul la fanioane eveniment:

— setare fanion eveniment;
— ștergere fanion eveniment;
— citire fanion eveniment;
— așteptare pentru un singur fanion eveniment;
— așteptare pe funcții SAU de mai multe fanioane;
— așteptare pe funcții ȘI de mai multe fanioane.

Pentru fanioanele eveniment din grupa celor comune (registrele 2 și 3) se mai furnizează încă 3 servicii:

— asociere fanion eveniment;
— dezasociere fanioane eveniment;
— ștergere fanion eveniment comun.

Cutii de scrisori. Fanioanele eveniment permit sincronizarea între procese, dar nu și comunicarea de date între acestea. În acest scop, VMS furnizează facilități de comunicare între procese prin cutii de scrisori. O cutie de scrisori este un periferic virtual care poate fi folosit la comunicarea între procese. El poate fi citit sau scris în același mod ca perifericele sau fișierele.

Unul din procese va insera mesaje la un capăt, iar celălalt va citi la altul; ca urmare, accesul se va face după algoritmul FIFO. Un proces care va citi cutia de scrisori va primi un mesaj sau o informație că aceasta este goală. Procesele pot cere de asemenea să fie informate atunci cînd se plasează un mesaj în cutie.

Creatorul unei cutii va specifica numărul maxim de mesaje pe care aceasta le va putea reține și dimensiunea maximă a unui mesaj. Cutiile de scrisori sînt protejate în același mod ca și fișierele VMS.

Zone de memorie comună. Cel mai general mecanism de comunicație între procese este prin zone comune de memorie. Procesele pot declara anumite *secțiuni globale* care vor fi zone de memorie folosite în comun de mai multe procese. Sincronizarea rămîne în seama proceselor care în acest scop pot recurge, de exemplu, la fanioane eveniment. Secțiunile globale pot fi permanente în sistem sau create dinamic de procese.

Fișiere partajate.

Procesele pot comunica și prin fișiere partajate. De exemplu, fișierele disc pot fi citite (dar nu și scrise) de oricîte procese. Dacă fișierele sînt secvențiale, ele pot fi accesate de către un singur proces care scrie și de mai multe care citesc. Dacă fișierele sînt cu acces direct ele pot fi utilizate în comun de către mai multe procese care scriu sau citesc din ele.

3.3. FIȘIERE ȘI DIRECTOARE

Un *fișier* este un grup de articole logice memorate pe un periferic de tip disc sau bandă. Un fișier poate conține un program, date necesare execuției unui program, o imagine grafică sau o documentație.

VMS, la fel ca alte multe sisteme de operare, recunoaște un tip special de fișier, numit *director*. Un director reprezintă repertoarul unor fișiere grupate în aceeași entitate logică. Directoarele și fișierele alcătuiesc o structură arborescentă pe volume, așa cum se poate vedea în fig. 3.11. Nodurile marcate cu triunghi reprezintă directoare.

Se poate observa că într-un director pot exista atît fișiere, cît și directoare; numărul maxim de niveluri ale arborelui este de 8. Structura fișierelor este de tip FILES-11, astfel încît orice volum creat sub RSX poate fi utilizat sub VMS (nu și invers).

Catalogul principal este identificat prin UIC [0,0]. Celelalte cataloage (utilizator) se identifică fie prin [nume], fie prin UIC: [g, m].

3.3.1 Specificarea unui fișier

Pentru a determina în mod unic un fișier el trebuie să primească un identificator unic. Utilizarea doar a numelui de fișiere în scopul precizării acestuia nu este suficientă, deoarece mai mulți utilizatori pot da din întîmplare același nume unor fișiere diferite. De aceea, în vederea localizării unui fișier se folosește un *specificator de fișier*. Acesta cuprinde nodul din rețea, perifericul unde rezidă fișierul,

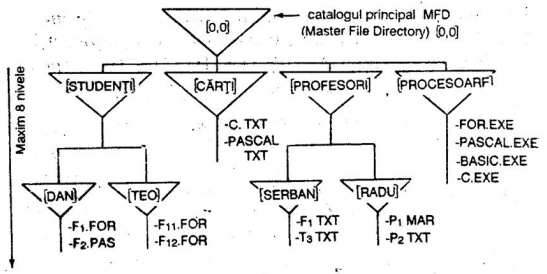


Fig. 3.11. Exemplu de structură de fișiere și directoare.

tipul, versiunea și desigur numele fișierului. Cu aceste informații VMS va localiza fișierul oriunde în rețea (cînd calculatorul reprezintă un nod DECnet)

Un exemplu de specificator de fișier complet este prezentat în fig. 3.12. În multe cazuri se pot folosi valori implicite pentru cîmpurile de nod, periferic și nume de director.

Semnificația cîmpurilor specificatorului este următoarea:

— *numele nodului* identifică calculatorul într-o rețea. Fiecare sistem din rețea este numit nod și se identifică printr-un nume unic, format din maxim 6 caractere și terminat prin caracterele :: Dacă nu se specifică în mod explicit un nume de nod, VMS va presupune că fișierul este localizat pe nodul local.

— *numele perifericului* reprezintă numele unei unități de disc sau de bandă magnetică, unde este memorat fișierul. Este format din 4 caractere alfanumerice care identifică tipul perifericului (primele 2 caractere), controlorul la care este acesta atașat (1 caracter) și numărul unității fizice (2 caractere). Numele perifericului se termină cu caracterul: Dacă se omite numele perifericului, VMS va presupune că se referă la un fișier de pe perifericul implicit al utilizatorului respectiv.

Dintre cele mai importante coduri de periferice precizăm: LB — imprimanta; MB — periferic cutie poștală; MU, MT, MB, MF, MS — diferite tipuri de bandă magnetică; NL — perifericul nul; OP — consola operator; RT — terminal interactiv; DA, DB, DD, DJ, DL, DU — pentru diferite tipuri de discuri; DX, DY — disc flexibil;

Fig. 3.12. Specificator de fișier.

— *directorul* specifică numele directorului care conține fișierul. Numele directoarelor sînt închise între paranteze drepte ([,]). Pentru precizarea directorului se poate folosi o cale care pornește din rădăcină. Dacă directorul nu este precizat, VMS va considera că este vorba de directorul implicit, introdus într-un fișier de administratorul sistemului;

— *numele fișierului* poate fi format din maxim 39 de caractere alfanumerice;

— *tipul fișierului*, format din caractere alfanumerice, indică tipul informației memorată în fișier. Putem introduce orice combinație de caractere drept tip de fișiere, dar o serie de utilitare și programe sistem folosesc următoarele tipuri implicite: BAS — program BASIC; COM — procedură de comandă; DAT — fișier de date de intrare; DIR — subdirector; DIS — listă de distribuție a poștei; EDT — fișier de inițializare editor EDT; EXE — imagine executabilă creată de link-editor; FOR — program FORTRAN; HLP — fișier „help"; LIS — fișier listă creat de compilator; MAC — program MACRO; OBJ — module obiect create de compilator; OLB — bibliotecă de module obiect; TLB — bibliotecă text; TMP — fișier tampon; TXT — fișier text;

— *numărul de versiune* este format din unul pînă la cinci caractere numerice. Cînd se creează un fișier el primește implicit versiunea 1. Ori de cîte ori fișierul este editat el primește un număr de versiune implementat cu 1. Dacă nu se specifică versiunea se consideră implicit 1 la creare sau cea mai mare la consultare. Se poate specifica și un număr de versiune negativ care indică a cîta versiune, pornind cu ultima înapoi, este cea referită. Se poate folosi 0 ca număr de versiune cu semnificația „ultima versiune sau -1 cu semnificația „cea mai veche versiune".

În specificarea unui fișier se pot folosi o serie de *caractere speciale* care reduc timpul de tastare, în special cînd referim mai multe fișiere sau directoare. Dintre acestea amintim :

„*" — poate fi utilizat pentru substituirea cîmpurilor nume de fișier și/sau versiune. Ele indică faptul că în acel

cîmp se acceptă orice combinație de caractere. De exemplu, F1.* indică toate fișierele cu nume F1 și cu orice extensie. În acest fel se pot desemna mai ușor grupe de fișiere, fără a fi nevoiți să tastăm numele fiecărui fișier;

„%" — poate fi utilizat la substituirea unui caracter individual indicînd că în acea poziție se poate accepta orice caracter. De exemplu, F%. FOR va indica faptul că este vorba fie de F1.FOR, fie de F2.FOR etc. Și în acest mod se pot specifica mai ușor, în cadrul comenzilor DCL, anumite grupe de fișiere;

„..." — se folosește pentru precizarea directorului, permițînd căutarea în jos în ierarhia de directoare. De exemplu, [CATEDRA...] F1. FOR va localiza fișierul F1. FOR, căutîndu-l în directorul CATEDRA și în subdirectoarele acestuia. În specificatorul [... *]. BAS se precizează toate fișierele de tip BAS din directorul implicit și din toate subdirectoarele sale;

„—" permite ca localizarea (căutarea) fișierului să se facă pînă la directorul indicat. De exemplu, [CATEDRA PROFESORI] F10. DAT face ca F10.DAT să fie căutat în directorul CATEDRA, începînd cu directorul PROFESORI.

3.3.2 Tipuri de fișiere

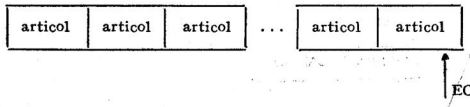
Sistemul de I/E cuprinde module pe mai multe niveluri: RMS, rutine executive, drivere de periferice, ACP (excluzînd acele discuri montate cu comanda MOUNT).

Sistemul de operare VMS furnizează o gamă de servicii RMS (Record Management Services) care sînt folosite pentru programarea intrărilor/ieșirilor într-un mod mai complex decît prin directiva QIO (la nivel executiv). RMS sînt orientate către articole logice, gestionînd blocarea și deblocarea acestora în blocuri fizice.

RMS poate crea și gestiona 3 *tipuri de fișiere*: secvențiale, relative, indexate.

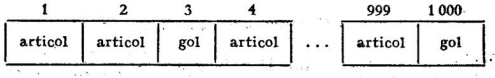
Ca urmare, se furnizează 3 *moduri de acces*: secvențial, prin cheie (aleatoriu), prin adresa de articol.

1) *Fișiercle secvențiale* au un mod simplu de organizare, articolele fiind citite sau scrise în ordine consecutivă (fig. 3.13).



2) *Fișierele relative* constau dintr-un ansamblu de celule de dimensiune fixă. Utilizatorul este cel care specifică dimensiunea celulei. Acestea sînt numerotate secvențial de la 1 pentru primul articol, pînă la ultimul. Fiecare celulă poate conține un singur articol sau poate fi goală. Celulele ocupate sau cele goale pot apărea oriunde în fișier; ele nu trebuie să fie contigue. Numerele celulelor sînt unice, fiind folosite pentru identificarea articolelor memorate în ele.

Un număr de celulă utilizat pentru identificarea unui articol se numește *număr relativ de articol* (fig. 3.14).



Organizarea relativă permite atît accesul secvențial, cît și aleatoriu, însă duce la o utilizare necorespunzătoare a spațiului pe disc.

3) *Fișierele secvențial-indexate* sînt necesare în proiectarea unor aplicații, în care localizarea în fișier trebuie să se facă cu viteză mare.

Poziționarea în fișier nu se face în funcție de numărul de articole, ci în funcție de datele conținute în articolul respectiv (fig. 3.15). Fiecare articol are un *cîmp cheie*, ale cărui poziție și lungime sînt fixe. RMS utilizează 5 tipuri de chei posibile : șir de caractere, întreg cu semn pe 16 biți, întreg fără semn pe 16 biți, întreg cu semn pe 32 biți, întreg fără semn pe 32 biți și zecimal împachetat.

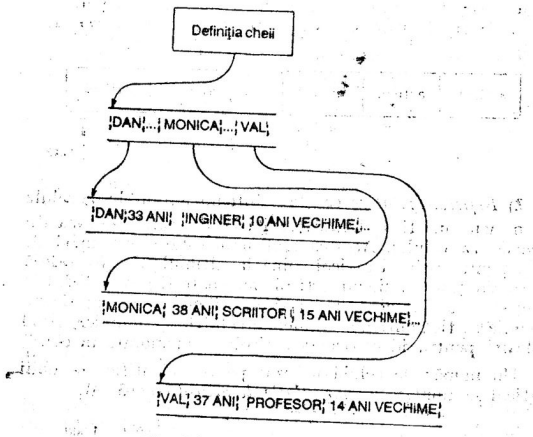


Fig. 3.13. Organizare secvențială.

RMS va construi o tabelă arborescentă, numită ***index***, formată din intrări care conțin valoarea cheii pentru articolele scrise în fișier și un pointer la acea locație a fișierului care conține articolul cu cheia respectivă. Se pot construi indexuri diferite pentru accesul la același fișier.

Organizarea secvențial-indexată permite accesul direct (aleatoriu), respectiv secvențial. De asemenea, este posibilă exploatarea acestor fișiere simultan de către mai mulți utilizatori atît la scriere, cît și la citire (fișierele relative pot fi doar scrise în acces partajat).

Vom analiza acum cele trei *moduri de acces* posibile în RMS.

1) *Accesul secvențial* permite citirea/scrierea articolelor în secvență. La fișierele cu organizare relativă accesul secvențial ignoră golurile între articole. Citiri succesive furnizează articole succesive, sărind peste celulele neocupate. Accesul secvențial la fișiere secvențial-indexate presupune parcurgerea articolelor în ordinea în care apar în index.

2) *Accesul aleatoriu* (direct) este posibil la toate tipurile de organizare. La fișierele secvențiale accesul aleatoriu se face doar dacă articolele au lungime fixă. La cererea de citire/scriere trebuie precizat numărul relativ de articol. Accesul direct la fișierele relative este însoțit de precizarea numărului relativ al articolului dorit. Dacă se încearcă citirea unei celule goale, RMS returnează un indicator de neexistență a articolului. Accesul aleatoriu la fișierele secvențial-indexate se face prin cheie; RMS caută cheia în index și cu pointerul găsit aici accede articolul căutat.

3) *Accesul prin adresa de articol* poate fi utilizat cu orice organizare de fișier, cu condiția să se cunoască adresa disc a articolului. După fiecare citire/scriere încheiată cu succes RMS furnizează adresa articol programului care a inițiat cererea de I/E. Programul va memora această adresă pentru utilizări ulterioare. Acest mod de acces duce la o mare viteză de prelucrare a fișierelor, dar cere un efort suplimentar de programare.

3.4. INTERFAȚA UTILIZATOR. LIMBAJUL DCL

3.4.1 Inițierea unei sesiuni

După conectarea terminalului la VAX/VMS se apasă tasta ⟨RETURN⟩, ⟨CTRL/C⟩ sau ⟨CTRL/Y⟩. Se va afișa mesajul:

WELCOME TO VAXIN
USER NAME:

Se va tasta numele, care trebuie în prealabil declarat administratorului de sistem; acesta îl va face cunoscut sistemului de operare. Apoi se cere parola:

PASSWORD:

Dacă parola și numele sînt corecte (corespund cu cele din UAF — *U*ser *A*utorization *F*ile), se afișează un mesaj de conectare, după care apare promterul ($) al limbajului

DCL (*D*igital *C*ommand *L*anguage). El indică faptul că sistemul este gata să accepte comenzi interactive. Se observă însă că „$" este promterul implicit; putem redefini alt promter cu comanda:

SET PROMT.

În cazul în care se dorește conectarea prin rețeaua DECnet la un alt calculator, după desfășurarea inițializării sesiunii de mai sus, se va da o comandă:

$ SET HOST nume-nod

care va face atașarea la calculatorul „nume-nod" situat la distanță. Apoi se vor putea executa comenzi DCL, ca orice utilizator local al acelui sistem. În paralel se va folosi comanda:

SHOW NETWORK pentru a se obține lista nodurilor din rețea.

Terminarea sesiunii de lucru se face cu comanda:

$ LOGOUT

Dacă la ieșirea din sesiune se folosește:

$ LOGOUT/FULL

se furnizează o serie de informații statistice:

— numărul de I/E;

— resurse folosite;

— numărul de întreruperi de pagină etc.

În cazul conectării în rețea, LOGOUT va încheia sesiunea de lucru la distanță, revenind în sesiunea locală inițiată mai întîi.

Comanda LOGOUT va încheia și sesiunea locală.

3.4.2 Limbajul de comandă DCL

Pentru a lucra cu VAX/VMS trebuie să utilizăm un limbaj de comandă, numit aici DCL (*D*igital *C*ommand *L*anguage), care va indica calculatorului acțiunile pe care să le întreprindă. DCL este un limbaj interpretativ, ale cărui comenzi, numite și *VERBE*, sînt în număr de aproximativ 200. Aceste comenzi posedă în plus o serie de *PARAMETRI* și *CALIFICATORI*.

Sintaxa generală a unei comenzi DCL este următoarea:

$ ETICHETA: NUME-COMANDA PARAMETRI/
/CALIFICATORI! COMENTARIU

ETICHETA — este opțională, formată din maxim 255 caractere. Ea este folosită în proceduri de comandă. Este separată de restul comenzii prin caracterul „:".

NUME-COMANDA — sau verbul, definește acțiunea care este cerută sistemului de operare.

PARAMETRII — specifică un obiect sau o listă de obiecte conform cărora trebuie să acționeze verbul. Dacă se precizează mai mulți parametri, ei trebuie separați cel puțin printr-un blanc.

CALIFICATORII — definesc sau modifică funcția DCL care trebuie executată. Ei constau din cuvinte cheie, eventual urmate de valori. Cuvintele cheie ale calificatorilor trebuie precedate de caracterul „/". Există 3 categorii de calificatori: parametri; poziționali; comenzi.

Calificatorii pot fi de următoarele forme:

/NUME

/NUME = VALOARE

/NUME = (VALOARE 1,..., VALOAREn)

Comenzile se pot introduce fie cu litere mari, fie cu litere mici, fie mixt. Interpretorul DCL le va converti automat în litere mari.

Comenzile DCL pot consta în 9 sau chiar mai multe caractere. De aceea, se poate folosi o *ABREVIERE* a lor, cu lungime mai mică. De exemplu, în loc de comanda DIRECTORY se poate folosi doar DIR. Dacă abrevierea folosită creează confuzii, interpretorul va cere mai multe caractere.

În cadrul comenzilor se pot folosi nume de periferice, directori, fișiere IMPLICITE, care nu mai trebuie precizate.

Comenzile DCL cer de obicei unul sau mai mulți PARAMETRI. Aceștia pot fi precizați direct în linia de comandă; dacă însă nu sînt precizați, sistemul cere valoarea fiecăruia din ei. De exemplu, comanda COPY are 2 parametri:

fișierul sursă și cel destinație. Ca urmare, putem tasta toți parametrii:

```
$ COPY F1.TXT F2.TXT
```

sau putem să-i furnizăm la cererea sistemului:

```
$ COPY
$_FROM:F1.TXT
$_TO:F2.TXT
```

Dacă în loc de un parametru furnizăm o listă de parametri, aceștia trebuie separați prin virgule, ca în exemplul:

```
$ PRINT F1.TXT,F2.TXT
```

În ceea ce privește *CALIFICATORII*, aceștia sînt de 3 tipuri: de comandă, de poziție și de parametri. Codificatorii de comandă se aplică întregii comenzi. În general, ei sînt plasați la sfîrșitul comenzii. Un calificator pozițional are diferite sensuri în funcție de locul unde este plasat în cadrul comenzii. De exemplu, dacă este plasat după verb, dar înaintea primului parametru va afecta întreaga comandă. Același parametru plasat după un parametru, îl va afecta numai pe acesta.

De exemplu,

```
$ PRINT / COPIES=3 F1.TXT, F2.TXT
```

va face cîte 3 copii la tipărirea fișierelor din listă.

În schimb comanda:

```
$ PRINT F1.TXT/COPIES=3,F2.TXT
```

va duce la listarea în 3 copii doar a fișierului F1.TXT. Calificatorii de parametri vor afecta doar parametrul care urmează. De exemplu:

```
$ PRINT F1.TXT/QUEUE=Q1,F2.TXT/QUEUE=
    =F2
```

va introduce F1 în firul de așteptare Q1, iar F2 în firul Q2.

Unii calificatori au valori numerice pozitive sau negative. Alții sînt formați dintr-un singur nume care poate fi prefixat eventual de „No" pentru a-l nega.

Fiecare comandă este analizată de INTERPRETORUL DCL, care în caz de eroare furnizează un mesaj de forma:

```
%FACILITATE—G—IDENT, TEXT
```

FACILITATE — reprezintă componenta VAX/VMS care este sursa erorii

—G— reprezintă gravitatea erorii (nivelul):

- S terminare cu succes
- I informațională
- W atenție
- E eroare
- F eroare fatală

IDENT — REPREZINTĂ ABREVIEREA TEXTULUI

TEXT — reprezintă explicația erorii

Una din facilitățile interesante ale limbajului DCL este posibilitatea de a reapela și reedita linii cu comenzi deja introduse. DCL memorează ultimele 20 de comenzi introduse la fiecare terminal. Pentru a reapela o comandă anterioară se folosește ⟨CTRL/B⟩. Pentru a vizualiza toate comenzile din buffer se folosește comanda:

```
$REÇALL/ALL
```

Pentru a vizualiza o anumită comandă (de exemplu a 2-a) se folosește comanda:

```
$ RECALL 2
```

Odată apelată comanda, se pot folosi tastele de editare pentru a o modifica și a o lansa din nou în execuție.

3.4.2.1 *Comenzi DCL referitoare la fișiere*

1) *CREATE*

FUNCȚIA: creează un fișier cu textul introdus pe perifericul SYS$INPUT, care este în mod obișnuit tastatura. Ca parametru va cere numele fișierului. Apoi comanda va aștepta introducerea unor linii pe care le va memora în fișierul nou creat. Cu ⟨CTRL/Z⟩ se încheie introducerea de date în fișier.

SINTAXA:

CREATE specificator—fișier

Calificatori:

- /LOG
- /OWNER-UIC
- /PROTECTION
- /VOLUME

Spre exemplu, comanda următoare creează un fișier de date:

```
$ CREATE CATALOG.DAT
POP
DAN
VASILE
 <CTRL/Z >
```

2) *CREATE/FDL*

FUNCȚIA: permite apelarea utilitarului FDL (*F*ile *D*efinition *L*anguage) care, pe baza specificațiilor din fișierul de definiții, va crea un nou fișier (fișier vid).

SINTAXA:

CREATE/FDL = fișier — specificații fișier

De exemplu:

```
$CREATE/PROTECTION=(S:RWED,0; RWED,
G, 14) F1.TXT
```

va crea fișierul F1.TXT, afectînd drepturi depline la sistem (S) și proprietar (0) și nici un drept la grup (G) și alți utilizatori (W).

3) *RENAME*

FUNCȚIA: definește un nou nume pentru un fișier și opțional îl realocă altui director

SINTAXA:

RENAME fișier — intrare[, ...] fișier ieșire

Calificatori:

/BACKUP	/BEFORE	/CONFIRM	/CREATED
/EXCLUDE	/EXPIRED	/LOG	/MODIFIED
/NEW-VERSION		/SINCE	

De exemplu, vrem ca fișierul CATALOG.DAT să primească numele GRUPA.DAT și să fie copiat într-un nou director:

```
$ RENAME CATALOG.DAT [INSTITUT.FACULTA-
TE] GRUPA.DAT
```

De notat că un fișier nu poate primi un alt nume și să fie copiat pe un alt volum.

4) *COPY*

FUNCȚIA: creează un nou fișier din conținutul unuia sau mai multor fișiere.

SINTAXA:

COPY fișier-intrare [,...] fișier ieșire

Calificatori:

/ALLOCATION	/BACKUP	/BEFORE	/CONCATENATE
/CONFIRM	/CREATED	/EXLUDE	/CONTIGUOUS
/EXPIRED	/LOG	/MODIFIED	/PROTECTION
/REPLACE	/SINCE	/VOLUME	REAE-CHECK
/WRITE-CHECK			

Comanda COPY este folosită pentru:

- — copierea unui fișier de intrare în unul de ieșire, opțional cu schimbarea numelui și a amplasării;
- — copierea unui grup de fișiere de intrare într-un grup de fișiere de ieșire;
- — concatenarea mai multor fișiere de intrare în unul de ieșire.

De exemplu, comanda:

```
$ COPY F1.TXT,F2.TXT F.TXT
```

va concatena fișierele F1 și F2 într-un fișier F.

5) *APPEND*

FUNCȚIA: permite adăugarea unuia sau mai multor fișiere de intrare la sfîrșitul unui fișier de ieșire.

SINTAXA:

APPEND fișier-intrare [,...] fișier-ieșire

Calificatori:

/ALLOCATION	/BACKUP	/BEFORE	/CONFIRM
/CONTIGUOUS	/CREATED	/EXCLUDE	/EXPIRED

/LOG /MODIFIED /SINCE /READ-CHECK
/WRITE-CHECK

6) *DELETE*

FUNCŢIA: permite ştergerea unui fişier sau unui grup de fişiere. Trebuie specificate numele fişierului, tipul şi versiunea în mod obligatoriu. Dacă în loc de numărul de versiune se foloseşte caracterul „;", se va şterge ultima versiune. De asemenea, în locul cîmpului nume, tip sau versiune se poate folosi caracterul „*" cu semnificaţia „toate combinaţiile găsite".

SINTAXA:

DELETE fişier [,...]

Calificatori:

/BACKUP /BEFORE /CONFIRM /CREATED /ERNSE
/EXCLUDE /EXPIRED /LOG /MODIFIED /SINCE

De exemplu, comanda:

$ DELETE F1.*;*

va şterge toate fişierele F1, indiferent de tip şi versiune.

Comanda:

$ DELETE/CONFIRM *.FOR;*

va căuta toate fişierele de tip FOR din directorul implicit şi la fiecare din ele, cu confirmarea utilizatorului, va face ştergerea.

Pentru utilizatorii care au fişiere confidenţiale, este indicat ca ştergerea acestora să se facă cu calificatorul /ERASE, care nu numai că va scoate fişierul din director, dar va scrie peste vechiile informaţii cu caracter aleator.

7) *TYPE*

FUNCŢIA: permite afişarea conţinutului unor fişiere text la terminal. Dacă fişierul care va fi afişat este de dimensiuni mari, se poate folosi calificatorul /PAGE, ceea ce va antrena scoaterea lui din *pagina în pagină*.

SINTAXA:

TYPE fişier [,...]

Calificatori:

/BACKUP /BEFORE /CONFIRM /CREATED /EXCLUDE
/EXPIRED /MODIFIED /OUTPUT /PAGE /SINCE

De exemplu,

$ TYPE/PAGE F2.TXT

8) *PRINT*

FUNCŢIA: realizează conuri tipărite la imprimantă ale unui fişier. În VMS fişierele sînt expediate direct la periferic de tipărire după execuţia lui PRINT. Ele sînt introduse într-un fir de aşteptare. Sistemul afişează un mesaj informaţional pe display, care indică numele fişierului, numele firului de aşteptare şi numărul intrării din fir. Apoi scrierea poate continua, fişierul urmînd a fi tipărit ulterior.

SINTAXA:

PRINT fişier [,...]

Calificatori:

Se poate folosi un număr foarte mare de calificatori, dintre care amintim: /NOTIFY care cere sistemului să anunţe printr-un mesaj cînd se termină tipărirea fişierului, /COPIES care indică sistemului cîte copii să fie făcute la imprimantă şi /QUEUE, calificator care permite introducerea fişierului într-un anumit fir de aşteptare, la un anumit terminal, cu facilităţi de editare cerute.

9) *PURGE*

FUNCŢIA: şterge toate fişierele cu acelaşi nume, mai puţin ultima versiune.

SINTAXA:

PURGE [fişier [,...]]

Calificatori

/BACKUP /BEFORE /CONFIRM /CREATED /ERASE
/EXCLUDE /EXPIRED /KEEP /LOG /MODIFIED
/SINCE

De exemplu, comanda:

$PURGE

va şterge toate fişierele care au mai multe versiuni, lăsînd la fiecare doar ultima versiune. Un exemplu în care se vor şterge toate fişierele cu sufixul DAT, mai puţin ultimele versiuni, este următorul:

$ PURGE *.DAT

Dacă se doreşte să se reţină mai mult decît ultima versiune, se foloseşte calificatorul KEEP;

$PURGE/KEEP=2

comanda ducînd la ştergerea tuturor fişierelor care au mai multe versiuni, mai puţin ultimele două versiuni.

10) DIFFERENCES

FUNCŢIA: compară conţinutul a două fişiere şi creează un listing cu acele articole care nu sînt identice.

SINTAXA:

DIFFERENCES fişier-principal fişier-revizuit

Calificatori:

/MATCH /MODE /PARALEL /NUMBER
/WINDOW /IGNORE /OUTPUT /MAXIMUM-DIFFERENCES

11) SORT

FUNCŢIA: permite apelarea utilitarului SORT care asigură sortarea unui fişier după un anumit criteriu.

SINTAXA:

SORT fişier-intrare [, ...] fişier-ieşire

12) MERGE

FUNCŢIA: permite apelarea utilitarului SORT care asigură combinarea a pînă la 10 fişiere sortate în unul singur. Fişierele de intrare trebuie în paralel sortate înainte de a apela comanda MERGE.

SINTAXA:

MERGE fişier-intrare1, fişier-intrare2 [, ...] fişier-ieşire

Calificatori:

/CHECK-SEQUENCE /KEY /STATISTICS
/INDEXED-SEQUENTIAL /SEQUENTIAL /SPECIFICATION
/FORMAT /CONTIGUOUS /DUPLICATES
/ALLOCATION /FORMAT /RELATIVE

13) SEARCH

FUNCŢIA: caută în unul sau mai multe fişiere anumite şiruri de caractere şi listează toate liniile care conţin apariţii ale acestor şiruri.

SINTAXA:

SEARCH fişier [,...] şir-caracter [,...]

Calificatori:

/EXACT /EXCLUDE /FORMAT /HEADING /LOG
/MATCH /NUMBERS /OUTPUT /STATISTICS /WINDOW

14) READ

FUNCŢIA: Citeşte un articol de pe fişierul de intrare specificat de numele-logic şi atribuie valoarea citită numelui-simbolic.

SINTAXA:

READ nume-logic : nume-simbolic

Calificatori:

/DELETE /MATCH /END OF FILE /INDEX /KEY
/MATCH /PROMPT /TIME-OUT

15) WRITE

FUNCȚIA: scrie un articol (expresie) pe fișierul de ieșire indicat de numele logic.

SINTAXA:

WRITE nume-logic expresie [, ...]

Calificatori:

/ERROR /UPDATE /SYMBOL

16) UNLOCK

FUNCȚIA: face un fișier accesibil, care devine inaccesibil datorită unei închideri incorecte.

SINTAXA:

UNLOCK fișier [, ...]

Calificatori:

/CONFIRM /LOG

17) BACKUP

FUNCȚIA: permite apelarea utilitarului BACKUP care asigură în principal copierea/restaurarea unor fișiere pe disc sau între disc și banda magnetică.

SINTAXA:

BACKUP fișier-intrare fișier-ieșire

18) DUMP

FUNCȚIA: permite afișarea conținutului unui fișier sau a volumului în reprezentare ASCII, zecimală, hexazecimală sau octală.

SINTAXA:

DUMP fișier [,...]

Calificatori:

/ALLOCATED /BLOCKS /BYTE /DECIMAL /FILE/HEADER
/FORMATED /HEADER /HEXA /LONGWORD /NUMBER
/OCTAL /OUTPUT /PRINTER /RECORDS /WORD

19) ANALYZE

FUNCȚIA: permite examinarea unui fișier sau a structurii discului

SINTAXA:

ANALYZE/calificator

Calificatori:

/DISK-STRUCTURE /RMS /IMAGE
/OBJECT /SYSTEM

3.4.2.2. *Comenzi DCL referitoare la directoare*

Administratorul sistemului creează pentru fiecare utilizator un director „rădăcină". Numele acestui director ca și numele unității pe care este instalat discul sînt memorate într-un fișier sistem special. În acest fel sistemul de operare știe care sînt fișierele fiecărui utilizator.

În acest director se pot crea alte subdirectoare în care pot fi incluse fișiere sau alte directoare. În felul acesta fiecare utilizator își poate crea un subarbore de fișiere legat prin directorul „rădăcină", putînd avea cel mult 7 niveluri de subdirectoare.

Pentru fiecare subdirector se creează de fapt un fișier director (cu extensia .DIR) în directorul curent. Se cuvine să se sublinieze că fișierele director sînt folosite exclusiv de către sistem pentru a cataloga conținutul directorului. Vom prezenta principalele comenzi cu care se poate crea, șterge sau vizualiza directorul.

1) CREATE/DIR

FUNCȚIA: crecază un nou director sau subdirector în vederea catalogării de fișiere.

SINTAXA:

CREATE/DIR director [,...]

Calificatori:

/LOG /OWNER-UIC /PROTECTION /VOLUME

De exemplu, să presupunem că directorul rădăcină este [CLASA] și că dorim să creăm un subdirector cu numele [RADU]. Vom folosi comanda:

$ CREATE/DIR [RADU]

Conform convenţiilor, numele de directoare sînt încadrate între paranteze drepte, iar subdirectoarele sînt precedate de punct.

În momentul în care vom vizualiza conţinutul directorului [CLASA] (cu comanda DIR), subdirectorul [.RADU] va apărea ca un fişier: RADU. DIR; 1.

2) SET DEFAULT

FUNCŢIA: permite specificarea directorului implicit de lucru pentru utilizator. Eventual, acest director se poate afla pe un alt periferic, care devine periferic implicit.

SINTAXA:

SET DEFAULT [periferic:] director

De exemplu, dacă mă aflu în directorul [CLASA] şi doresc să lucrez în fişierele din directorul [.RADU], voi schimba directorul de lucru cu comanda:

$ SET DEFAULT [CLASA. RADU]

sau mai simplu:

$ SET DEFAULT [.RADU]

Ulterior, pentru a reveni în directorul [CLASA] putem folosi comanda:

$ SET DEFAULT [CLASA]

sau

$ SET DEFAULT [-]

3) DIR

FUNCŢIA: permite listarea tuturor fişierelor dintr-un director sau obţinerea de informaţii despre un fişier sau grup de fişiere

SINTAXA:

DIR [fişier[,...]]

Calificatori:

/ACL	/BACKUP	/BEFORE	/BY-OWNER	/CREATED
/DATE	/EXCLUDE	/EXPIRED	/FILE-ID	/FULL
/GRAND-TOTAL		/MODIFIED	/OUTPUT	/OWNER
/PROTECTION		/SECURITY	/SELECT	/SINCE
/SIZE	/TOTAL	/VERSIONS	/WIDTH	

De exemplu,

$ DIR/EXCLUDE=(*.EXE,*.FOR)

va afişa directorul implicit, mai puţin fişierele de tip EXE şi FOR

Comanda:

$ DIR [A...] *.FOR

va lista toate fişierele de tip FOR din directorul A şi din subdirectoarele sale.

Comanda:

$ DIR [*...]

va lista tot conţinutul discului

4) DELETE

Această comandă permite inclusiv ştergerea unor directoare.

În vederea ştergerii unui subdirector (de exemplu [.RADU] trebuie parcurse 4 etape.

a) Să se şteargă toate fişierele din subdirector;

$ SET DEFAULT [.RADU]

$ DELETE *.*;*

b) Să ne mutăm în directorul din nivelul imediat superior:

$ SET DEFAULT [CLASA]

c) Să se reseteze codul de protecţie al subdirectorului:

$ SET PROTECTION =(O:D) RADU.DIR;1

Resetarea protecţiei este necesară deoarece, implicit subdiectoarele sînt protejate împotriva ştergerii de către proprietar. Asupra codului de protecţie vom reveni pe larg.

d) Să se şteargă subdirectorul

$ DELETE RADU.DIR;1

3.4.2.3. *Comenzi DCL referitoare la volume*

1) MOUNT

FUNCȚIA: permite apelarea utilizatorului MOUNT care face ca un volum de disc sau bandă să fie accesibil la utilizare. Operația este necesară înainte de a lucra cu un volum sau fișier al acestuia. Volumul devine „on-line"

SINTAXA:

MOUNT nume-periferic [:] [,...] [et-volum [,...]]

Calificatori:

/ACCESSED	/AUTOMATIC	/BIND	/BLOCKSIZE	/CACHE
/CLUSTER	/COMMENT	/CONFIRM	/COPY	/DENSITY
/EXTENSION	/FOREIGN	/GROUP	/INIT	/LABEL
/MESSAGE	/MOUNT-VERIFICATION		/OWNER-UIC	/PROTECTION
/RECORDSIZE	/SHADOW	/SYSTEM	/UMLOAD	/WRITE

2) DISMOUNT

FUNCȚIA: permite demontarea unui volum disc sau bandă care a constituit inițial obiectul unei comenzi MOUNT

SINTAXA:

DISMOUNT nume-periferic [:]

Calificatori:

/ABORT /UNIT /UNLOAD /CLUSTER

3) INIȚIALIZE

FUNCȚIA: are rolul de a formata un disc și de a scrie eticheta acestuia. Formatarea presupune pierderea oricăror informații care existau anterior pe disc. Se creează cîteva fișiere standard, printre care catalogul principal -MFD [0,0], fișierul index care cuprinde headerele fișierelor etc.

SINTAXA:

INITIALIZE nume-periferic [:] eticheta-volum

Calificatori:

/ACCESSED	/BADBLOCKS	/DATA-CHECK	/DENSITY
/DIRECTORIES	/ERASE	/EXTENSION	/FILE-PROTECTION
/GROUP	/HEADERS	/INDEX	/LABEL
/MAXIMUM-FILES		/OWNER-UIC	/PROTECTION
/SHARE	/STRUCTURE	/SYSTEM	/USER-NAME
/VERIFIED	/WINDOWS		

3.4.2.4. *Comenzi DCL referitoare la nume logice*

NUMELE LOGICE reprezintă nume care substituie specificatorul unui fișier sau o porțiune dintr-un specificator de fișier. Ele au două funcții de bază: independență de fișier și periferic și, respectiv, scurtarea specificării fișierelor.

Independența de fișier și periferic înseamnă că utilizatorul nu este constrîns de un anumit periferic, de exemplu disc DUA1, operațiile de intrare/ieșire făcîndu-se de către un nume „logic" care poate fi asociat unui periferic.

Pentru a ilustra acest lucru, să presupunem spre exemplificare o bază de date mare PERSONAL.DAT în directorul [BAZE]. Directorul este pe un disc amovibil, care rezidă de obicei pe DUA2:. Toate programele care vor actualiza baza de date se află pe perifericul implicit al utilizatorului, DUA1: Deoarece programele și baza de date sînt în directoare diferite și pe periferice diferite, va trebui să se folosească în program specificatorul complet al bazei de date.

Să vedem ce se întîmplă dacă perifericul DUA1: se defectează. Pila trebuie mutată pe o altă unitate și programul trebuie modificat cu noul specificator. Dacă însă se folosește nume logic, soluția constă într-o simplă redirectare a numelui-logic la noul periferic fizic.

Dar putem folosi nume logice și atunci cînd specificatorul unui fișier este foarte lung, trecînd prin mai multe șiruri de subdirectoare. Vom putea asigna acest specificator (sau o porțiune din el) la un nume logic, care va putea fi tratat mult mai repede.

1) DEFINE

FUNCȚIA: creează nume logic și îi asignează șiruri de caractere sau liste de șiruri de caractere ca valoare.

SINTAXA:

DEFINE nume-logic șir-caractere [,...]

Calificatori:

/EXECUTIVE-MODE /GROUP /JOB /LOG
/NUME ATTRIBUTES /PROCES /SYSTEM /TABLE
/TRANSLATION-ATTRIBUTES /USER-MODE
/FORM /KEY

De exemplu,

$DEFINE FIS1 DUA2: [BAZE] PERSONAL.DAT

va face declararea numelui logic FIS1, căruia îi asignează șirul echivalent DUA2: [BAZE] PERSONAL.DAT. În continuare, ori de cite ori se referă acest fișier se poate folosi, în locul specificatorului, numele logic FIS1.

Dacă numele logic reprezintă un periferic bandă sau disc, numele acestuia trebuie terminat cu 2 puncte:

$ DEFINE DISC DUA1:

Prin nume logice se pot defini și părți din specificator. Ele trebuie să fie prima parte (stînga) și trebuie separate de rest prin caracterul „:", ca în exemplul:

$DEFINE FIS2 DISC: [BAZE] PERSONAL.DAT

Numele logice definite în acest mod pot fi folosite apoi în toate celelalte comenzi DCL

2) ASSIGN

FUNCȚIA: asociază un nume logic la un nume de periferic, specificator de fișier sau un alt nume logic.

SINTAXA:

ASSIGN nume-logic șir-caractere

3) ALLOCATE

FUNCȚIA: furnizează acces exclusiv la un periferic și, opțional, stabilește un număr logic pentru acest periferic. Dacă perifericul este alocat, el nu poate face obiectul unei noi alocări.

SINTAXA:

ALLOCATE nume-periferic: [nume-logic]

4) OPEN

FUNCȚIA: deschide un fișier pentru intrare/ieșire și asignează un nume logic fișierului, nume pe care-l plasează în tabela de nume logice a procesului.

SINTAXA:

OPEN nume-logic [:] fișier

5) CLOSE

FUNCȚIA: permite închiderea unui fișier deschis cu OPEN și dezasignarea numelui logic asociat fișierului.

SINTAXA:

CLOSE nume-logic

6) SHOW LOGICAL

FUNCȚIA: permite afișarea numelor logice și a șirurilor de caractere care le sînt asociate.

SINTAXA:

SHOW LOGICAL nume-logic [:] [,...]

Calificatori:

/ACCES – MODE /ALL /DESCENDANTES /FULL
/GROUP /JOB /OUTPUT /PROCESS
/STRUCTURE /SYSTEM /TABLE

Se indică unde se află NUMELE LOGIC: în tabela de nume asociată procesului, în cea asociată grupului sau în tabela de nume logice a sistemului.

7) *DEASSIGN*

FUNCȚIA: permite dezasignarea numelor logice care au făcut obiectul unor comenzi ASSIGN.

SINTAXA:

DEASSIGN nume—logic [:]

8) DEALLOCATE

FUNCȚIA: eliberează un periferic alocat prin comanda ALLOC, ștergînd și numele logic aferent.

SINTAXA:

DEALLOCATE periferic:

— NUME LOGICE SISTEM

VMS folosește o serie de nume logice proprii. Ele sînt utilizate în comenzile DCL, fiind necunoscute de interpretorul DCL. Numele logice sistem implicite pot fi identificate prin prezența unui caracter „$" și a prefixului „SYS" în fața numelui. Cele mai importante sînt următoarele:

SYS$COMMAND — reprezintă fișierul inițial de unde interpretorul DCL citește comenzi. El este asignat terminalului la care lucrează utilizatorul.

SYS$DISK — reprezintă discul implicit.

SYS$ERROR — reprezintă perifericul pe care afișează sistemul erorile; este asignat la terminal.

SYS$INPUT — reprezintă fișierul de pe care interpretorul DCL citește comenzi și date (implicit terminalul).

SYS$OUTPUT — reprezintă fișierul pe care interpretorul DCL sau programele scriu răspunsuri sau date.

Numele logice sînt memorate în fișiere sistem numite TABELE DE NUME LOGICE. Dintre acestea următoarele 4 sînt cele mai folosite:

— GROUP TABLE — care conține numele logice disponibile tuturor utilizatorilor care fac parte din grupul asociat UIC-ului;

— JOB TABLE — conține numele logice accesibile procesului și tuturor subproceselor create de acesta;

— PROCESS TABLE — conține numele logice accesibile doar unui proces;

— SYSTEM TABLE — conține numele logice folosite de toți utilizatorii.

3.4.2.5. *Comenzi DCL informaționale și de control al contextului utilizator*

Dacă administratorul (inginerul) de sistem va permite accesul la VMS, va trebui să declare sistemului de operare un CONT. Acesta este un identificator (nume) propriu unui anumit utilizator, prin care este recunoscut de sistem. La începutul sesiunii de lucru, fiecare utilizator se va identifica la terminal prin CONT și PAROLĂ. Aceasta din urmă este sub controlul propriu al utilizatorului și poate fi modificată atunci cînd se dorește.

Sistemul de operare, dacă va recunoaște numele furnizat drept cont și parolă, va aloca resursele și privilegiile utilizatorului pe baza unor informații memorate în FIȘIERUL UAF (*U*ser *A*utorization *F*ile). Acesta este un fișier sistem controlat și modificat doar de inginerul de sistem.

Există cîte un articol și informații de identificare pentru fiecare utilizator autorizat.

Sistemul de operare identifică utilizatorii și grupurile de utilizatori după CODUL DE IDENTIFICARE — UIC (*U*ser *I*dentificator *C*ode). El reprezintă „dovada" prin care procesele, directoarele, fișierele și celelalte obiecte sînt identificate de sistem. Un UIC poate fi specificat fie numeric, fie alfanumeric. Specificarea numerică este de forma:

[grup, membru]

De exemplu [200, 203] identifică grupul 200 și membrul 203, ambele numere fiind reprezentate în octal.

Un UIC alfanumeric conține numele membrului și, opțional, numele grupului:

[membru] sau [grup, membru]

De exemplu, [CLASA, RADU 1] identifică grupul CLASA și membrul RADU1.

În momentul creării unui articol în fișierul UAF, inginerul de sistem acordă utilizatorului anumite PRIVILEGII și RESTRICȚII.

RESURSE. Fiecare utilizator primește anumite limite pentru consumul de resurse (memorie, UC, periferice, numărul de fișiere simultan deschise, etc). Aceste limite se referă la procesul acelui utilizator și la toate subprocesele create de el.

PRIORITATEA PROCESULUI. Există 32 de niveluri de priorități, de la 0 la 31. Cea mai mare este 31. Prioritățile sînt împărțite în 2 categorii: pentru multiprogramare (0-15) și pentru timp real (16-31). Prioritatea implicită a utilizatorului este 4.

PRIVILEGII UTILIZATOR. Prin intermediul sistemului de privilegii se controlează obținerea unor performanțe superioare pentru sistemul de operare. VAX/VMS conține un număr de 35 de privilegii, împărțite în următoarele 7 categorii, clasificate după nivelul drepturilor posedate de sistem.

FARA — fără privilegiu;

NORMAL — minimum de privilegii necesare folosirii sistemului;

GROUP — privilegii care dau posibilitatea să devină membru în același grup;

MARI — potențial să consume toate resursele necritice ale sistemului;

SISTEM — facilități de inferență cu operațiile sistem normale;

FISIER — potențial de trecere peste sistemul de protecție al fișierului;

TOATE — posibilitatea de a controla toate funcțiile sistem.

RESTRICȚII DE ACCES. Acestea pot specifica zilele și orele cînd se permite acces local la sistem sau la distanță prin rețea DECnet. De asemenea, se pot aloca unui utilizator facilități de acces la fișiere sau comunicații inter—taskuri, fără să fie necesară furnizarea, în linia de comandă, a contului și parolei.

„COTA"—DE ACCES. La declararea unui utilizator în VMS, i se atribuie o „cotă" de disc care reprezintă limita maximă a spațiului disc (măsurată în blocuri de 512 octeți), pe care o pot ocupa fișierele sale. Aceste informații sînt memorate în fișierul sistem QUODA.SYS care este controlat de inginerul de sistem. În momentul atingerii „cotei", utilizatorul este atenționat printr-un mesaj de eroare. Se recomandă în astfel de situații să se folosească comanda PURGE pentru a șterge versiunile vechi ale fișierelor utilizator. Vom analiza principalele comenzi DCL care permit vizualizarea și/sau modificarea contextului de lucru al utilizatorului (proceselor).

1) *SHOW TERMINAL*

FUNCȚIA: afișează caracteristicile unui anumit terminal: viteză, tip, lungime rînd și pagină, tip de paritate etc.

SINTAXA:

SHOW TERMINAL [periferic [:]]

2) *SET TERMINAL*

FUNCȚIA: permite modificarea unor caracteristici ale terminalului.

SINTAXA:

SET TERMINAL [periferic [:]]

De exemplu, comanda:

```
$ SET TERMINAL/WIDTH=40
```

va determina ca lungimea unei linii să fie de 40 caractere. Dacă însă declarăm o lungime a liniei mai mare decît lățimea ecranului, se sare automat la linie nouă și se continuă scrierea pe nivelul următor cu comanda:

```
$ SET TERMINAL/NOWRAP
```

Trecerea la linie nouă se inhibă dacă linia este prea lungă. Se vor scrie toate caracterele în exces pe ultima poziție a liniei. Dacă pe terminal se execută o lucrare importantă care dorim să nu fie perturbată de primirea unor scrieri, se poate folosi comanda:

```
$ SET TERMINAL/NOBROADCAST
```

Atunci cînd am terminat şi dorim să autorizăm din nou operaţia pe ecran a unor mesaje de poştă, vom folosi opţiunea /BROADCAST:

$ SET TERMINAL/BROADCAST

Blocarea mesajelor de poştă se poate face cu comanda: SET BROADCAST

Una din cele mai utilizate facilităţi în VMS este posibilitatea reapelării unor linii de comandă mai vechi în vederea editării lor şi a refolosirii. Există 2 moduri de editare pe care le putem selecta cu comanda SET TERMINAL:

1) *MODUL INSERARE*, cînd caracterele tastate vor fi inserate la dreapta cursorului;

2) *MODUL ŞTERGERE*, cînd caracterele indicate de cursor vor fi şterse şi înlocuite de cele tastate. Comenzile folosite sînt:

$ SET TERMINAL/INSERT — pentru intrare în mod inserare;

$ SET TERMINAL/OVERSTIKE — pentru intrare în mod ştergere.

În oricare mod de editare am fi, putem trece dintr-un mod în altul cu ⟨CTRL/A ⟩

3) *DEFINE/KEY*

FUNCŢIA: permite să se asccieze la cheile programabile ale claviaturii comenzi DCL mai frecvent folosite. În acest fel anumite comenzi se pot tasta foarte repede, apăsînd doar tasta asociată.

SINTAXA:

DEFINE/KEY nume—cheie „sir—comanda—DCL"

De exemplu, dacă folosim des comanda PURGE pentru curăţirea din moment în moment a directorului propriu de fişierele vechi, putem să asociem această comandă tastei programabile PF1 astfel:

$ DEFINE/KEY PF1 „PURGE DUA1: [BAZE] *.*"

Se poate folosi calificatorul /TERMINATE asociat cu aceste comenzi, ceea ce va face ca odată apăsată tasta programată, să se şi lanseze execuţia comenzii DCL, fără a mai aştepta tastarea lui <RETURN>.

4) *SET AUDIT*

FUNCŢIA: permite activarea sau dezactivarea controalelor VMS de securitate.

SINTAXA:

SET AUDIT

Calificatori:

/ALARM /DISABLE /ENABLE

5) *SET DAY*

FUNCŢIA: permite setarea zilei implicite specificate în fişierul UAF pentru ziua curentă.

SINTAXA:

SET DAY

Calificatori:

/DEFAULT /LOG /PRIMARY /SECONDARY

6) *SET TIME*

FUNCŢIA: permite setarea orei sistem.

SINTAXA:

SET TIME [=ora]

7) *SET DEVICE*

FUNCŢIA: stabileşte imprimanta sau terminalul care joacă rol de „spooler".

SINTAXA:

SET DEVICE periferic [:]

Calificatori:

/AVAILABLE//SPOOLED /DUAL — PORT

8) *SET DIRECTORY*

FUNCŢIA: modifică caracteristicile directorului specificat.

SINTAXA:

SET DIRECTORY director [...]

Calificatori:

/BACUP/BEFORE /CONFIRM /CREATED /EXCLUDE /EXPIRED /MODIFIED /OWNER — UIC /SINCE /VERSION—LIMIT

9) *SET HOST*

FUNCŢIA: Conectează terminalul la care se lucrează, prin calculator, la un alt calculator din reţeaua DECnet.

SINTAXA:

SET HOST nume-mod

10) *SET KEY*

FUNCŢIA: schimbă starea de definiţie curentă a cheilor. Ele au fost definite cu comanda DEFINE /KEY.

SINTAXA:

SET KEY

Calificatori:

/LOG /STATE

11) *SET LOGINS*

FUNCŢIA: defineşte numărul utilizatorilor care pot avea acces la sistem. Comanda afişează nivelul de interactivitate curent.

SINTAXA:

SET LOGINS

Calificatori:

/INTERACTIVE

12) *SET MEGTAPE*

FUNCŢIA: defineşte caracteristicile importante care se asociază la un periferic bandă magnetică în vederea unor utilizări ulterioare.

SINTAXA:

SET MEGTAPE periferic [:]

Calificatori:

/DENSITY /END-OF-FILE /LOG /REWIND /SKIP /UNLOAD

13) *SET MESSAGE*

FUNCŢIA: permite specificarea formatului mesajelor sistem ori ignorarea lor.

SINTAXA:

SET MESSAGE [fişier]

Calificatori:

/DELETE /FACILITY /IDENTIFICATION /SEVERITY /TEXT

14) *SET PRINTER*

FUNCŢIA: permite stabilirea caracteristicilor imprimantei

SINTAXA:

SET PRINTER imprimanta [:]

Calificatori:

CR /FF /LA11 /LA180 /LOWERCASE /LP11 /PAGE
PASSALL /PRINTAK /TAB /TRUNCATE /UPPERCASE
/WIDTH /WRAP

15) *SET PROMPT*

FUNCŢIA: defineşte un nou prompter DCL. Cel implicit este caracterul „$". Pentru a face noul prompter permanent este bine să se includă comanda SET PROMPT în fişierul de iniţializare LOGIN.COM

SINTAXA:

SET PROMPT = [şir—caractere]

16) *SET UIC*

FUNCŢIA: permite stabilirea valorii implicite a UIC-ului utilizator.

SINTAXA:

SET UIC uic

17) *SET VOLUME*

FUNCŢIA: modifică caracteristicile unui volum FILES-11.

SINTAXA:

SET VOLUME periferic [:] [/...]

Calificatori:

/ACCESED /DATA-CHECK /EXTENSION /FILE-PROTECTION
/LABEL /LOG /MOUNT-VERIFICATION
/OWNER-UIC /PROTECTION /RETENTION /UNLOAD
/USER-NAME /WINDOWS

18) *SET WORKING-SET*

FUNCȚIA: permite setarea valorii implicite a dimensiunii zonei de memorie de lucru pentru un proces sau a limitei superioare pe care zona de memorie o poate atinge prin execuția imaginii unui proces.

SINTAXA:

SET WORKING—SET

Calificatori:

/ADJUST/EXTENT /LIMIT /LOG /QUOTA

19) *SHOW DEFAULT*

FUNCȚIA: permite afișarea perifericului și directorului implicite

SINTAXA: SHOW DEFAULT

20) *SHOW DEVICES*

FUNCȚIA: afișează starea perifericului din sistem. Produce o listă exhaustivă a tuturor perifericelor conectate la sistem sau a caracteristicilor unui anumit periferic

SINTAXA:

SHOW DEVICES [periferic [:]]

21) *SHOW KEY*

FUNCȚIA: afișează definiția cheilor programabile.

SINTAXA:

SHOW KEY [nume—cheie]

22) *SHOW MEGTAPE*

FUNCȚIA: afișează caracteristicile unei anumite unități de bandă

SINTAXA:

SHOW MEGTAPE

23) *SHOW MEMORY*

FUNCȚIA: afișează disponibilitatea și utilizarea memoriei.

SINTAXA:

SHOW MEMORY

Calificatori:

/ALL /FILES /FULL /OUTPUT /PHYSICAL — PAGES /POOL /SLOTS

24) *SHOW NETWORK*

FUNCȚIA: afișează informații despre rețeaua DECnet din care face parte calculatorul.

SINTAXA:

SHOW NETWORK

25) *SHOW PRINTER*

FUNCȚIA: afișează caracteristicile imprimantei.

SINTAXA:

SHOW PRINTER periferic [:]

26) *SHOW QUOTA*

FUNCȚIA: afișează informații legate de „cota" de disc alocată unui utilizator ca limită maximă de folosință.

SINTAXA:

SHOW QUOTA

27) *SHOW TERMINAL*

FUNCȚIA: afișează caracteristicile unui anumit terminal.

SINTAXA:

SHOW TERMINAL [PERIFERIC [:]]

28) *SHOW TIME*

FUNCȚIA: afișează data și ora curente.

29) *SHOW USERS*

FUNCȚIA: afișează lista utilizatorilor conectați în acel moment la sistem, numele terminalelor la care lucrează, conturile și identificatorii proceselor care se execută (pic).

30) *SHOW WORKING-SET*

FUNCȚIA: afișează limita curentă a zonei de lucru din memorie pentru procesul curent.

31) *HELP*

FUNCȚIA: apelează utilitarul HELP care afișează informații despre sintaxa unor comenzi DCL sau despre VMS.

SINTAXA:

HELP

Calificatori:
/INSTRUCTIONS /LIBLIST /LIBRARY /OUTPUT /PAGE /PROMPT /USERLIBRARY

3.4.2.6. *Comenzi DCL referitoare la procese*

1) *SHOW PROCESS*

FUNCȚIA: permite obținerea unor informații despre procesul principal și eventualele subprocese care au fost create. Se vor putea vedea numele procesului, prioritatea UIC-ul și specificatorul implicit de fișere.

SINTAXA:

SHOW PROCESS [nume proces]

Calificatori:

/ALL /DETERMINATION /MEMORY /PRIVILEGES /QUOTAS /SUBPROCESSES

2) *SET PROCESS*

FUNCȚIA: permite modificarea unor caracteristici ale unui proces. Implicit se consideră procesul curent.

SINTAXA:

SET PROCESS [nume-proces]

Calificatori:

/CPU /DUMP /IDENTIFICATION /NAME /PRIORITY /PRIVILEGES /RESUME /SUSPEND /SWAPPING

De exemplu, comanda:

$SET PRIVILEGES / NAME = „DAN"

va redefini numele procesului la „DAN".

3) *SHOW STATUS*

FUNCȚIA: afișează informații despre procesul curent.

4) *SHOW SYSTEM*

FUNCȚIA: afișează informații despre toate procesele active în sistem.

SINTAXA:

SHOW SYSTEM

Calificatori:

/BATCH /FULL /NETWORK /OUTPUT /PROCESS /SUBPROCESS

5) *RUN*

FUNCȚIA: plasează o imagine în execuție în cadrul unui proces sau creează un subproces pentru a lansa în execuție o imagine specificată.

6) *STOP*

FUNCȚIA: specifică numele unui proces care va fi șters din sistem.

SINTAXA:

STOP [nume-proces]

7) *SYNCHRONIZE*

FUNCȚIA: plasează procesul care execută comanda în starea de așteptare pînă cînd un anumit program termină execuția.

SINTAXA:

SYNCHRONIZE [nume-program]

8) *WAIT*

FUNCȚIA: plasează procesul curent în starea de așteptare pînă cînd trece o anumită cuantă de timp.

SINTAXA:

WAIT cuanta-timp

9) *DEPOSIT*

FUNCȚIA: permite scrierea directă la anumite locații ale memoriei virtuale. Este utilizată în conjuncție cu EXAMINE, în timpul depanării interactive a unor programe.

SINTAXA:

DEPOSIT adresa = data [,...]

Calificatori:

/ASCII /BYTE /DECIMAL /HEXADECIMAL /OCTAL /WORD

10) *EXAMINE*

FUNCȚIA: permite vizualizarea conținutului memoriei virtuale.

SINTAXA:

EXAMINE adresa [: adresa]

Calificatori: aceiași ca la DEPOSIT.

11) *SPAWN*

FUNCȚIA: permite crearea unor subprocese. Acestea reprezintă una din facilitățile esențiale ale sistemului de operare VMS, ducînd la ridicarea gradului de multiplicare. Multiplicarea nu este limitată numai la posibilitatea execuției simultane a mai multor programe lansate la terminale diferite; utilizatorii pot crea procese multiple, numite subprocese, din procesul principal.

SINTAXA:

SPAWN [șir-comanda]

Calificatori:

/CLI /INPUT /KEYPAS /LOG /LOGICAL-NAMES
/NOTIFY /OUTPUT /PROCESS /PROMPT /SYMBOLS
/TABLE /WAIT

Implicit, SPAWN creează un subproces cu atributele procesului „tată" (director implicit, privilegii, „cota", disc, etc.). De exemplu, comanda:

$SPAWN

va face ca procesul „tată" să se pună „în hibernare" și subprocesul creat să primească controlul tastaturii. Utilizatorul poate executa orice comandă DCL; pentru a se reîntoarce la procesul „tată" se folosește comanda:

$LOGOUT

Comanda SPAWN afișează o serie de mesaje care pot fi inhibate prin calificatorul /NOLOG.

Se pot executa direct comenzi DCL sau alte programe (imagini) prin intermediul lui SPAWN, introducînd în continuare comanda corectă din punct de vedere sintactic. În general, este vorba de taskuri care se pot lansa și executa fără intervenția operatorului (de exemplu, procese de compilare). În comanda:

$SPAWN / NOWAIT / NOLOG RUN KERMIT

se cere execuția în paralel (/NONWAIT) a taskului KERMIT concurent cu procesul „tată" care poate cere execuția altor comenzi.

12 *ATTACH*

FUNCȚIA: permite transferul controlului de la un proces la un subproces creat sau invers. Comanda dă posibilitatea „conectării" tastaturii la un anumit proces. Ea reprezintă o metodă alternativă de a ieși dintr-un subproces.

SINTAXA:

ATTACH [nume-proces]

Să presupunem că vrem să creăm un subproces din procesul RADU. Se va executa o comandă:

$SPAWN

DCl va afișa mesajul:

%DCL-S-SPAWNED process RADU-1 spawned

care indică crearea subprocesului RADU-1, ce va primi controlul asupra tastaturii. Cînd dorim ieșirea din subprocesul RADU-1 și afectarea tastaturii la procesul RADU, executăm comanda

$ATTACH RADU

%DCL-S-RETURNED control returned to process RADU

Atunci cînd se iese cu ATTACH dintr-un subproces, acesta va intra în „hibernare". Ulterior, se poate atașa din nou tastatura acestui subproces și lucrul poate continua.

Observație. Orice proces poate fi întrerupt în vederea creării unui subproces. Acest lucru se poate face cu ⟨CTRL/Y⟩, ceea ce produce întreruperea procesului activ. Apoi, cu SPAWN se creează noul proces. Comanda:

$CONTINUE

permite apoi reluarea procesului întrerupt cu ⟨CTRL/Y⟩.

3.4.2.7. *Comenzi DCL referitoare la securitate*

Securitatea calculatoarelor a devenit deosebit de importantă în sistemele cu acces multiplu sau în rețele. VAX/VMS oferă o serie de facilități destinate să realizeze o creștere a securității și integrității contextului de execuție al utilizatorilor.

Elementul central al *protecției fișierelor* îl constituie *codul utilizator* — UIC (*U*ser *I*dentificator *C*ode). Acești utilizatori cu interese comune pot alcătui un grup; ei au anumite drepturi comune asupra resurselor, însă accesul între conturi poate fi permis sau interzis în funcție de structura de UIC-uri pe care o stabilește administratorul sistemului și de privilegiile atribuite conturilor individuale în fișierul UAF (*U*ser *A*uthorization *F*ile).

Pentru introducerea contului unui utilizator, inginerul de sistem are la dispoziție programul AUTHORIZE. El permite declararea sau modificarea numelui utilizatorului, parolei, UIC-ului, perifericului implicit. Se pot folosi 2 comenzi ale programului: ADD (pentru adăugări utilizator) și MOD (pentru modificare caracteristici utilizator):

```
$RUN AUTHORIZE
UAF > ADDnume / PASS = parole / UIC = [g,m] /
DEV = periferic / DIR = director
```

sau

```
UFD > MOD nume / PASS = parola / UIC = [g, m]/
DEV = periferic / DIR [director]
```

Sistemul de protecție se bazează pe *împărțirea utilizatorilor în 4 categorii*:

SYSTEM(S) — administratorul de sistem;
OWNER(O) — proprietarul contului;
GROUP (G) — utilizatori în același grup cu proprietarul contului;
WORLD(W) — toți ceilalți utilizatori din sistem.

Există de asemenea *4 tipuri de accese la fișiere*:

READ(R) — dreptul de citire a unui fișier;
WRITE (W) — dreptul de a crea sau modifica un fișier;
EXECUTE(E) — dreptul de a executa un proces;
DELETE (D) — dreptul de a șterge un fișier.

Cu acest sistem oricărei categorii de utilizatori i se poate permite sau interzice accesul la un fișier. Există însă două situații de excepție.

În primul rînd, orice persoană care are drept de EXECUȚIE poate citi un fișier dintr-un subdirector, dacă îi cunoaște numele și tipul.

În al doilea rînd, dacă privilegiile SYSTEM includ posibilitatea de a trece peste orice protecție, oricine din afara categoriei SYSTEM poate citi orice fișier.

Controlul accesului este capacitatea unui utilizator de a schimba condițiile de protecție la o resursă. Este de fapt un al cincilea tip de acces care poate fi specificat în lista de control al accesului (ACC). În VMS acest drept este conferit la două categorii: SYTEM și OWNER

În momentul creării unui fișier, el primește următoarele valori implicite de protecție:

SYSTEM : RWED
OWNER : RWED
GROUP : RE
WORLD : fără acces

Administratorul poate schimba aceste valori implicite pentru a spori nivelul de protecție.

1) *SHOW PROTECTION*

Funcția: afișează protecția implicită care se alocă fișierelor nou create în timpul sesiunii. Se indică pentru fiecare din cele 4 categorii de utilizatori care sînt afectate în mod implicit.

2) *SET PROTECTION*

FUNCȚIA: modifică protecția unui anumit fișier sau grup de fișiere. Ea este necesară atunci cînd dorim să ridicăm nivelul securității anumitor fișiere.

Sintaxa:

```
SET PROTECTION = [(cod) fișier ,...]
```

Calificatori:

```
/CONFIRM/LOG / PROTECTION / DEFAULT / DEVICE
```

De exemplu, comanda:

```
$SET PROTECTION = (O : RWE, G,W) FISH1.
COM
```

va afecta fișierului FIS1 drepturile R, W, E pentru proprietar, iar utilizatorilor din grup(g) și rest(w) nu li se va permite nici un drept de acces. Drepturile sistem(s) vor rmîne neschimbate.

Trebuie să observăm însă că VMS evaluează mai întîi protecția directorului și abia apoi a fișierului care se cere accesat. Ca urmare, chiar dacă avem dreptul, de exemplu, de citire la un fișier, dacă nu avem același drept și asupra directorului, fișierul nu este inaccesibil.

Vom sublinia de asemenea că este inutil să se retragă din drepturile SYSTEM (administratorului); acesta poate trece peste sistemul de protecție și în plus ștergerea drepturilor SYSTEM face imposibilă copierea sau salvarea fișierelor cu utilitarul BACKUP.

Vom prezenta cîteva sugestii utile în legătură cu protecția:

— Comanda SET PROTECTION este foarte utilă; dacă vrem să protejăm anumite fișiere importante împotriva ștergerii, se vor retrage drepturile de ștergere (D) la toate categoriile (O, W, G);

— Este indicat să se șteargă drepturile de acces la WORLD (W) pentru toate directoarele și subdirectoarele care conțin date senzitive;

— După ce am editat un fișier cu EDT va trebui să folosim o comandă SET PROTECTION pentru a repune vechea protecție la noua versiune de fișier; altfel, EDT pune protecția implicită;

— Pentru a permite unui alt utilizator să execute un fișier imagine (fișier. EXE), dar să nu-l poată copia în alt director, se va aloca drept de EXECUȚIE (E), dar se va retrage dreptul de citire, READ (R), la acel fișier.

2) *SET PASSWORD*

Funcția: permite schimbarea parolei de către proprietarul contului. Parola reprezintă „prima linie" împotriva unor accese neautorizate.

SINTAXA:

SET PASSWORD

Calificatori:

/GENERATE / SECONDARY / SYSTEM

Cînd se apelează comanda SET PASSWORD, sistemul cere mai întîi vechea parolă. Dacă aceasta este furnizată

corect, se cere noua parolă, care o va înlocui pe cea veche. pentru verificare, se mai cere o dată noua parolă.

```
$SET PASSWORD
old password: (tastez vechea parolă)
New password: (tastez noua parolă)
Verification: (tastez din nou noua parolă)
```

Parola poate fi formată din maxim 31 de caractere alfanumerice, inclusiv caracterul de subliniere (—). Unele implementări cer ca parola să fie mai mare de 6 caractere.

Pentru a alege o parolă mai „obscură" și greu de dedus pentru un utilizator neautorizat, se recomandă să se utilizeze facilitatea VMS de *generare de parolă*. Se vor genera la cerere 5 parole aleatoare, din care utilizatorul poate alege una:

```
$SET PASSWORD / GENERATE
old password:
labiunlu        la-bi-un-lu
jeyuzil         jeyu-zil
ilumio          i-lu-mio
olhyzi          ol-hy-zi
laceput         la-ce-put
chase a new password from this list, or press return to
get a new list
New password:
Verification:
```

3.5. EDITORUL DE TEXTE EDT

Editorul standard al sistemului de operare VMS este EDT. Cu ajutorul lui utilizatorii pot crea fișiere de date, fișiere cu programe sursă, fișiere cu documentații sau corespondență. De asemenea, se pot face ștergeri, inserări, corecții sau rearanjări ale liniilor unor astfel de fișiere text.

Trebuie menționat faptul că EDT permite lucrul în trei moduri, dintre care două vor fi discutate în acest paragraf. De asemenea, EDT furnizează facilități de redefinire a unor taste și de creare a unor fișiere de comenzi de inițializare. Aceste fișiere permit stabilirea la apelarea EDT a caracteristicilor de editare dorite.

Editorul EDT are și facilități de reluare în caz de eroare. În timpul editării el creează și actualizează un „fișier jurnal", care va memora toate comenzile ce au fost date în timpul sesiunii. El va putea fi folosit în eventualitatea unei blocări de sistem pentru refacerea textului editat.

3.5.1. Apelarea EDT

Apelarea editorului de texte se face cu comanda:

$EDT nume-fișier

sau

$EDIT/EDT nume-fișier

În acest moment EDT va stabili un spațiu de lucru pentru fișier, numit *buffer de lucru principal* (MAIN). De asemenea, este deschis și un fișier „jurnal". În mod tipic, cele două buffere ocupă 80 de blocuri disc fiecare; ca urmare, trebuie să existe un minim de 160 locuri libere în contul propriu pentru a putea utiliza EDT.

Dacă fișierul indicat nu există, se va crea unul nou; EDT va afișa mesajul:

Input file does not exist

Asteriscul care urmează este prompter-ul editorului EDT și indică faptul că se așteaptă o comandă EDT.

Dacă se editează un fișier existent, el nu este modificat de EDT. Acesta creează o copie a fișierului în bufferul MAIN, unde se vor opera toate modificările asupra textului. De abia la sfîrșitul sesiunii, (comanda EXIT) copia din fișierul MAIN va deveni un fișier util care reprezintă versiunea modificată a fișierului inițial.

3.5.2. Moduri de lucru cu EDT

EDT poate fi utilizat în trei moduri: *linie*, *caracter* și, respectiv, *necaracter*.

Modul linie se utilizează de obicei pentru terminale de tipărire. El constă dintr-un număr de comenzi utilizate pentru schimbarea unor linii din textul original.

Modul caracter este folosit pe videoterminale de tip VT 100, VT 200. El presupune mutarea cursorului în interiorul unei „ferestre" de 22 de linii, care este afișată pe ecranul terminalului. Cursorul poate fi mutat la orice locație a ferestrei. Acest lucru permite inserarea, schimbarea sau ștergerea de text oriunde pe ecran.

Modul necaracter permite editarea unor texte pe videoterminale care dispun de o tastatură numerică. În acest caz textul apare în partea superioară a ecranului, iar comenzile de editare sînt afișate în partea de jos a ecranului. Se va tasta ⟨ RETURN ⟩ pentru apelarea unei anumite comenzi.

3.5.2.1. *Modul linie*

După apelul EDT, editorul intră automat în modul de lucru linie. Acesta este indicat prin apariția prompterului*. Fiecare comandă linie se dă după asterisc și determină cu ⟨ RETURN ⟩. Liniile fișierului sînt numerotate secvențial de către editor. Ele pot fi adresate individual sau pe grupe prin numere de linie sau domenii de numere de linie. Vom prezenta în continuare principalele comenzi care pot fi folosite în mod linie, semnificația lor și, în paranteze, abrevierile care pot fi folosite pentru ele:

*CHANGE (C) — permite trecerea din modul de linie în modul caracter;

*COPY (CO) — permite copierea unui grup de linii dintr-o porțiune a textului în alta. De exemplu, * CO 5:10 to 20 permite copierea liniilor de la 5 la 10 imediat înaintea liniei 20;

*DELETE (D) — permite ștergerea unei linii sau grup de linii;

*EXIT (EX) — termină sesiunea EDT, salvează fișierul modificat, creînd o nouă versiune;

*HELP (H) — afișează toate comenzile mod linie și informațiile asupra lor;

*INCLUDE (INC) — permite copierea unor fișiere text extern și includerea lor în fișierul de editat;

*INSERT (I) — permite inserarea unor linii înaintea poziției curente din fișier. Ieșirea din modul inserare se face cu ⟨ CTRL/Z ⟩;

*MOVE (M) — permite mutarea unor linii dintr-un loc în altul al fişierului;

*QUIT (QUIT) — permite ieşirea din EDT fără salvarea buffer-ului MAIN în care s-au operat modificări;

*REPLACE (R) — permite înlocuirea unei linii sau a unui grup de linii cu una sau mai multe linii care se introduc în modul INSERT;

*RESEQUENCE (RES) — face numerotarea liniilor din fişier;

*SUBSTITUTE (S) — face substituirea unui şir de caractere cu un altul. Permite efectuarea unei corecţii în cadrul liniilor. Şirul vechi şi cel nou sînt încadrate între paranteze ascuţite (⟨, ⟩):

S/ ⟨ text-vechi ⟩/ ⟨ text-ncu ⟩;

*WRITE (WR) — permite scrierea textului din buffer într-un nou fişier. De exemplu,

*WR F1.TXT 60 : 80

face să se creeze fişierul F1.TXT, în care să se scrie toate liniile cuprinse între 60 şi 80.

3.5.2.2. *Modul caracter*

Modul caracter furnizează facilităţi de inserare, ştergere, copiere, substituţie a unor diverse entităţi ale textului sursă (caractere, cuvinte, linii, paragrafe, pagini, propoziţii şi buffere).

Este un mod de lucru orientat pe ecrane, deoarece se poate edita text pe orice porţiune a ecranului.

Intrarea în modul caracter se face cu comanda „CHANGE". Odată intrat în acest mod de lucru, se afişează un ecran format din 22 de linii text. Dacă apare pe ecran mesajul [EOB], acesta indică sfîrşitul bufferului.

Inserarea de text se face simplu, tastînd caracterele dorite. În momentul cînd se apasă <RETURN>, se consideră că în acel loc este sfîrşitul unei linii. Pentru manipularea liniilor şi a textului conţinut în ele se folosesc tastele numerice din partea dreaptă a tastaturii terminalelor de tip VT 52, VT 100, VT 200.

La toate aceste tipuri de terminale, apăsînd tasta ⟨ HELP ⟩, se va putea obţine afişarea pe ecran a amplasării tastelor.

Tastele utilizate pentru modul caracter sînt cele suplimentare, existente în partea dreaptă a tastaturii. Ele sînt taste numerice şi de mutare a cursorului. Fiecare din aceste taste, exceptînd ⟨ GOLD ⟩ şi ⟨ HELP ⟩, au cîte 2 funcţii. Pentru a alege cea de-a doua funcţie se va apăsa tasta ⟨ GOLD ⟩ împreună cu tasta dorită.

În fig. 3.16 se indică structura acestei tastaturi suplimentare pentru terminalele de tip VT 100. În comenzile modului caracter, care se vor prezenta în continuare, se va indica rolul fiecăreia din aceste taste.

ADVANCE — setează cursorul în modul său de lucru implicit: deplasarea înainte cu un caracter, cuvînt, linie sau altă entitate. Este utilizată pentru revenirea din modul BACKUP;

APPEND — permite adăugarea de text selectat cu comanda SELECT. Se va selecta zona de text dorită cu SELECT, apoi apăsînd pe tasta APPEND, acest text se va adăuga la sfîrşitul bufferului;

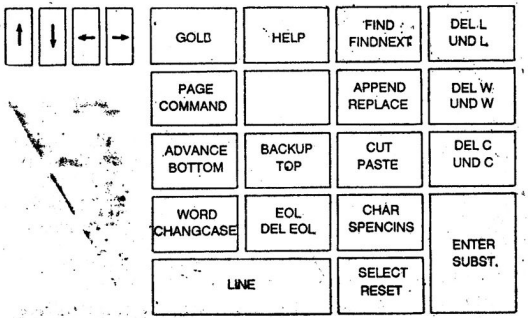


Fig. 3.16. **Tastatură utilizată în modul caracter.**

BACKUP — setează deplasarea cursorului înapoi;

BUTTOM — mută cursorul la sfîrșitul fișierului editat (va apărea simultan pe ⟨ GOLD ⟩ și pe ⟨ BUTTOM ⟩);

CHANGCASE — va comuta din caractere mici în caractere mari sau invers;

COMAND — permite intrarea în modul de lucru linie pentru a furniza o comandă; apare prompterul „COMAND";

CUT — reprezintă, împreună cu PASTE, cea mai puternică comandă EDT. Cu ajutoru lor se rearanjează textul. Folosindu-se CUT, se mută o porțiune de text, selectată cu SELECT, în bufferul auxiliar, de unde va putea fi luată cu ajutorul lui PASTE;

DELETE — permite ștergerea unor caractere (DEL C), linii (DEL L), cuvinte (DEL W) sau „end of line" (DEL EOL). Ștergerea se va aplica entității de după sau de dinaintea cursorului, după cum se lucrează în modul ADVANCE sau BACKUP. Ultima entitate ștearsă poate fi readusă în text cu ajutorul comenzii UNDELETE;

EOL — permite mutarea cursorului la sfîrșitul liniei curente;

FIND — permite căutarea unui anumit șir de caractere, cerut utilizatorului. Căutarea se face înainte (dacă se lucrează cu ADVANCE) sau înapoi (dacă se lucrează în BACKUP);

HELP — permite apelarea facilității de HELP a editorului;

LINE — permite mutarea cursorului la începutul liniei următoare;

PASTE — permita plasarea conținutului bufferului auxiliar în bufferul MAIN, începînd cu poziția curentă a cursorului. Se folosește în conjuncție cu comanda CUT;

REPLACE — permite ștergerea unei porțiuni de text selectate cu SELECT și înlocuirea ei cu conținutul bufferului auxiliar;

RESET — setează editorul în modul implicit (ADVANCE), anulează setările făcute cu SELECT;

SELECT — permite selectarea unor porțiuni de text, scop în care se va pune cursorul la începutul zonei și se va

apăsa tasta SELECT. Apoi se va muta cursorul pînă la sfîrșitul zonei. În continuare, se va putea aplica altă comandă, ca de exemplu CUT;

SPECINS — permite introducerea în text a unor caractere speciale;

SUBSTITUTE — permite înlocuirea unui text cu cel din bufferul auxiliar. Zona selectată cu SELECT va fi înlocuită cu conținutul acestui buffer. De exemplu, vrem să corectăm într-un text cuvîntul „RED" cu „READ". Se va folosi în acest scop secvența de comenzi:

⟨ SELECT ⟩ READ ⟨CUT ⟩ ⟨ GOLD ⟩ ⟨ FIND ⟩ VEX ⟨ SUBS ⟩

TOP — permite mutarea cursorului la începutul fișierului editat;

UNDELETE — permite refacerea ultimului caracter (UND C), cuvînt (UND W) sau linii (UND L) șterse cu comanda DELETE;

WORD — permite mutarea cursorului cu un cuvînt înainte sau înapoi;

CTRL/Z — permite ieșirea în modul linie; reapare prompterul „*". Apoi se poate ieși din EDT cu comanda EXIT.

3.5.3. Generarea manualului EDT

Beneficiarul poate să-și creeze un manual propriu cu informații despre EDT prin copierea informațiilor din fișierul HELP al EDT și apoi tipărirea lor. Se va face mai întîi asignarea:

```
*ASSIGN EDT. TXT SYS$OUTPUT
```

Apoi se va intra în modul de linie al EDT și se va tasta „HELP":

```
*HELP .....
```

Ca urmare a acestei comenzi, conținutul lui HELP se va copia în fișierul EDT. TXT. Apoi se va ieși din EDT cu EXIT și se va comanda DCL:

```
$ASSIGN TTnn: SYS$OUTPUT
```

unde TTnn este numărul terminalului utilizatorului. Apoi putem tipări fișierul EDT.TXT

3.5.4. Fişierul jurnal

Din momentul activării EDT, acesta activează un *fişier jurnal* unde memorează în mod secvenţial toate comenzile pe care le-a dat utilizatorul. Cînd se iese din editare în mod normal, se crează noua versiune cu fişierul text, iar fişierul jurnal este şters. Dacă însă sistemul se blochează în timpul lucrului, fişierul jurnal nu se distruge. Ca urmare, se poate reface fişierul text la starea de dinainte de blocare. lansînd sistemul EDT astfel:

$EDIT/EDT/RECOVER fişier

În acest caz, editorul va folosi fişierul jurnal pentru a face toate operaţiile de modificare a textului sursă în vederea aducerii sale la ultima formă premergătoare blocării. Apoi se poate continua editarea fişierului.

3.5.5. Editorul orientat pe sintaxa LSEDIT

În VMS, în afara editorului standard EDT, mai există şi un *editor sensibil la sintaxă* — LSEDIT —, care pe lîngă editarea textului face şi o verificare sintactică a instrucţiunilor introduse. Lansarea lui se face astfel:

$LSEDIT /LANGUAGE = limbaj fişier,

unde *limbaj* indică FORTRAN, COBOL, C etc. Acest calificator poate lipsi, editorul deducînd tipul limbajului după extensia dată fişierului (.FOR, .COB, .C etc.).

În cadrul programelor există două categorii de simboluri:

terminale — cele care încep cu litere mari şi fac parte din limbaj (de exemplu, în FORTRAN, SUBROUTINE etc.);

neterminale — care se înlocuiesc cu elemente de limbaj construite de programator.

Atît simbolurile terminale, cît şi cele neterminale pot apărea ca obligatorii ... [sau opţionale] ... .

Pentru manipularea simbolurilor se pot folosi următoarele *comenzi*:

1) EXPAND sau (CTRL/E) — care aplicată unui neterminal face descompunerea (expandarea) lui în unităţi sintactice componente;

2) ERASE PLACEHOLDER/FORMAT sau (CTRL/K) — care determină ştergerea unui neterminal; ca urmare, terminarea unei liste se face cu CTRL/K;

3) GOTO PLACEHOLDER/FORWARD sau (CTRL/N) — face trecerea la neterminalul următor;

4) GOTO PLACEHOLDER/REVERSED sau (CTRL/P) — face trecerea la neterminalul premergător.

Editorul are două moduri de lucu:

— *comandă* — cînd, în partea de jos a ecranului, se pot da comenzi precum COMPILE (compilare program) sau SPAWN (creare subprocese);

— *ecran* — pentru editare.

Trecerea în modul comandă se face cu (CTRL/Z), iar revenirea în mod ecran cu (CTRL/E)

Vom exemplifica utilizarea editorului LSEDIT pentru scrierea unui program FORTRAN

Se face lansarea editorului:

$LSEDIT PROG.FOR

Fiind vorba de un fişier nou, se afişează

[program unit]
[end of file]

Se va poziţiona cursorul pe [program unit], iar apoi se va tasta EXPAND (CTRL/E).

Pe ecran, în partea de jos, se vor afişa posibilităţile de continuare:

MAIN program unit
SUBROUTINE program unit
FUNCTION program unit
BLOCK DATA program unit
choise or press HELP

Apoi, folosind tastele de poziţionare a cursorului, se va selecta una din opţiunile de mai sus. Ca urmare, pe ecran, ce era în fereastră, [program unit] se înlocuieşte cu sintaxa opţiunii.

De exemplu, dacă am selectat o unitate de tip SUBROUTINE se va afişa:

```
SUBROUTINE  nume [ ({p} ...)]
  [subroutine — coments]
  [implicit — statements]
```

[executable — statements]
RETURN
END

Cu ajutorul comenzii EXPAND (CTRL/E) se detaliază un terminal, iar cu DESTROY (CTRL/K) se indică faptul că acel neterminal nu mai interesează (de exemplu, comentariile).

Cu comenzile CTRL/N și CTRL/P, operatorul se plasează pe anumite neterminale înainte sau înapoi.

Printre comenzile necunoscute de LSEDIT se numără și REVIEW, care după comanda COMPILE va face poziționarea pe erorile detectate. Cu comenzile CTRL/F se va avansa la următoarea eroare, iar cu CTRL/B la cea anterioară.

3.6. PROCEDURI DE COMANDĂ

Procedurile de comandă reprezintă facilități de utilizare superioară a sistemului de operare VMS. În cadrul lor este posibilă asignarea unor întregi comenzi la un singur cuvînt; manipularea de șiruri și caractere sau numere, precum și obținerea unor informații privind contextul de execuție a utilizatorului.

3.6.1. Simboluri

Un simbol este un *nume* care este ales pentru a reprezenta un șir de caractere, o valoare numerică sau logică. Numele (simbolurile) pot avea de la 1 la 255 de caractere și trebuie să înceapă cu un caracter, semnul „$" sau „—".

Crearea simbolurilor se face prin asignări de forma:

nume-simbol = valoare (pentru simboluri locale)
nume-simbol == valoare (pentru simboluri globale)

Un simbol local poate fi folosit la nivelul la care este definit. De exemplu, dacă se definește un simbol local și unul global într-o procedură de comandă, cînd procedura se termină, simbolul local va fi șters pe cînd cel global va rămîne.

De exemplu,

$LATIME = „SET TERMINAL/WIDTH = 132"

va asigura simbolului local „LATIME" comanda „DCL SET TERMINAL/WIDTH = 132". Din acel moment, de cîte ori se va tasta LATIME la nivelul unei comenzi DCL se va executa comanda SET. În acest exemplu se ilustrează una din importantele aplicații ale simbolurilor; reducerea volumului de caractere care trebuie tastate.

$REZULTAT = 18 × 127 × (10 — 2)

Simbolul REZULTAT va conține rezultatul operațiilor aritmetice indicate. Pentru a afișa conținutul unui simbol se poate folosi comanda DCL SHOW SYMPAC:

$SHOW SYMBOL REZULTAT

Pentru a lista simboluri globale, în comanda de mai sus trebuie folosit calificatorul /GLOBAL. Pentru listarea tuturor simbolurilor se folosește calificatorul /ALL.

Ștergerea simbolurilor se face folosindu-se comanda DELETE SYMBOL, ca în exemplul:

$DELETE SYMBOL REZULTAT

Și în această comandă se pot folosi calificatorii /GLOBAL ȘI /ALL.

Substituirea simbolurilor cu valoarea lor se face automat în timpul execuției comenzilor DCL în următoarele situații:

— simbolurile de la începutul comenzilor;
— simbolurile din funcțiile lexicale;
— simbolurile din instrucțiunile WRITE și IF;
— simbolurile din dreapta semnelor de asignare (=, ==).

În celelalte cazuri, utilizatorul poate forța substituirea simbolurilor prin punerea lor între apostrofuri, ca în exemplul:

$EU = „DJA1 : [BAZA] FIS.DAT"

$DIRECTORY „EU"

În interiorul unui șir de caractere, substituirea se poate forța încadrînd simbolul între ghilimele:

$ȘTERGE = „PURGE"„EU"

Cea mai importantă aplicație a simbolurilor este aceea de a produce mnemonice, care apoi să faciliteze tastarea unor comenzi DCL frecvent folosite. O altă aplicație im-

portantă a simbolurilor este *execuția unor proceduri de comandă*, ca în exemplul următor:

$ REȚEA = „CSYS $ LOGIN : REȚEA"

unde simbolul indică execuția unei proceduri de comandă. De asemenea, se pot crea comenzi noi cu ajutorul simbolurilor. În exemplul următor se va defini simbolul KERMIT, care va fi utilizat pentru execuția programului KERMIT. EXE localizat în directorul SYS$SYSTEM:

$KERMIT :== RUN SYS$SYSTEM : KERMIT

Inginerul de sistem poate defini pentru uz general o serie de simboluri care să reprezinte comenzi mai des folosite.

3.6.2. Reprezentarea datelor și expresiilor

Datele sînt memorate în următoarele moduri:

— *bit* — o cifră de 0 sau 1;
— *byte* (octet) — format din 8 biți;
— *cuvînt* — format din 16 biți;
— *cuvînt lung* — format din 32 de biți;
— *șir de caractere* — format din serii de 8 biți, fiecare grup reprezentînd codul ASCII al caracterului.

Expresiile DCL cuprind date și simboluri combinate cu următoarele categorii de operatori:

— *relaționali*: .GE, .LE., .NE., .GT., .LT. (pentru numere)
— *logici*: .AND., .OR., .NOT., (pentru șiruri de caracter)
— *aritmetici*: +, —, *,/

Expresiile pot fi:

— *logice*, a căror evaluare duce la un rezultat de (1 adevărat) sau 9 (fals);

aritmetice, a căror evaluare duce la obținerea unei valori numerice.

3.6.3. Funcții lexicale

Funcțiile lexicale sînt rutine, VMS care returnează informații despre procese sau sisteme ori permit manipularea unor date furnizate de utilizator. Funcțiile lexicale se utilizează ca niște simboluri, rezultatul apelării înlocuind numele funcției. Numele acestor simboluri reprezintă funcțiile lexicale și încep cu

F $ nume-funcție (parametru...)

De exemplu, funcția F$SEARCH returnează numele unui fișier furnizat ca parametru, după căutarea lui prealabilă în arbore. Dacă fișierul nu este găsit se returnează șirul nul. De exemplu,

```
$FIȘIER = F$SEARCH („DUA1 : [... [F1.DAT")
$SHOW SYMBOL FIȘIER
   FIȘIER=„DUA1 : [FACULTATE-CATEDRA1]
   F1, DAT; 1"
```

O alta funcție este F$EDIT care manipulează date, ca în exemplul:

```
$SIR = „șerban  george"
$SIR = F$EDIT (SIR, „COMPRESS, UPCASE")
$SHOW SYMBOL SIR
   SIR = „ȘERBAN  GEORGE"
```

Vom prezenta în continuare, pe scurt, *principalele funcții lexicale*, parametrii de apel și rolul lor:

**F$CVSI (poziție, lățime, șir)*

Este folosită pentru a extrage un cîmp de biți dintr-un șir de caractere. Rezultatul este convertit într-un întreg cu semn.

** F$CVTIME (timp-intrare, timp-ie,ire, cîmp)*

Convertește timpul în forma „yyyy-mm-dd hh:mm:ss.cc"

**F$CVUI (poziție-bit, lățime, număr)*

Extrage un cîmp de biți dintr-un șir de caractere și îl convertește la un întreg fără semn.

**F$DIRECTORY( )*

Returnează numele directorului implicit.

**F$EDIT (șir, lista-editare)*

Realizează editarea unui șir de caractere conform cu lista.

**F$ELEMENT (număr-element, delimitator, șir)*

Extrage un element dintr-un șir de caractere în care elementele sînt separate de delimitatorul indicat.

F$EXTRACT (şablon, lungime, şir)

Extrage un subşir dintr-un şir de caractere.

F$FILE ATTRIBUTES (fişier, articol)

Returnează informaţii privind atributele unui fişier.

F$GETDVI (periferic, articol)

Returnează informaţii de stare şi identificare despre sistemul local sau despre alt mod.

F$INTEGER (expresie)

Converteşte rezultatul unei expresii în întreg.

F$LENGTH (şir)

Returnează lungimea unui şir de caractere.

F$LOCATE (subşir, şir)

Returnează poziţia unui subşir în şirul de caractere.

F$MODE ( )

Returnează un şir de caractere care reprezintă modul în care se execută procesul.

F$PID (simbol)

Returnează identificatorul procesului (PID).

F$PROCESS( )

Obţine numele procesului curent.

F$SEARCH (fişier)

Caută în director şi returnează specificatorul complet dacă fişierul este găsit.

F$STRING (expresie)

Returnează şirul de caractere echivalent rezultatului expresiei.

F$TIME( )

Returnează data şi ora curentă ca şir de caractere.

F$TYPE (simbol)

Returnează tipul simbolului.

F$USER( )

Returnează codul de identificare (UIC) al utilizatorului curent.

3.6.4. Crearea procedurilor de comandă

O *procedură de comandă* este un fişier care conţine o secvenţă de comenzi DCL ce pot fi executate interactiv sau „batch". Procedurile de comandă sînt folosite în VMS pentru executarea unor sarcini repetitive sau complexe în vederea reducerii timpului necesar tastării repetate a unor comenzi.

De exemplu, se ştie că la intrare în sistem utilizatorul trebuie să facă cîteva operaţii pregătitoare: stabilirea directorului implicit, stabilirea funcţiei cheie PF1 care să reprezinte comanda DIRECTORY, curăţirea directorului de versiunile mai vechi ale fişierului, reţinînd de exemplu doar ultimele două versiuni.

Pentru a realiza aceste operaţii, trebuie tastate următoarele comenzi DCL:

```
$SET DEFAULT DUB1: [GRUPA.NOTE]
$DEFINE/KEY/TERMINATE PF1 „DIRECTORY/
SIZE"
PURGE/KEEP = 2/LOG
```

Această secvenţă de comenzi DCL, folosită frecvent, se poate cataloga sub forma unor proceduri de comandă într-un fişier , cu EDT sau cu comanda CREATE. Fişierul care conţine procedura de comandă trebuie să fie de tip „.COM". În continuare, ori de cîte ori se doreşte execuţia acelei secvenţe de comenzi, se apelează fişierul cu procedură astfel:

```
$ INIT.COM
```

unde „ω" indică faptul că fişierul conţine o procedură de comandă.

Procedurile de comandă pot fi formate din simple comenzi DCL, executate în secvenţă, dar pot conţine şi facilităţi mai complexe, cum ar fi simboluri, funcţii lexicale, etichete şi instrucţiuni de control al fluxului de intrare-ieşire. Ca urmare, aceste proceduri se pot constitui în programe suficient de puternice.

Crearea procedurilor de comandă se face în general cu editorul (EDT sau EVE). Fiecare comandă trebuie să înceapă cu caracterul „$", după care se pot lăsa mai multe spaţii

pentru aranjarea programului. Comenzile pot fi continuate pe mai multe linii; acest lucru se indică prin caracterul „—" la sfîrșit de linie, ca în exemplul următor:

```
$SET TERMINAL —
      /ADVANCED VIDEO —
      /WIDTH = 80. —
      /PAGE = 20.
$ — — —
```

În procedura de comandă se pot introduce și linii cu date, care nu sînt prelucrate de interpretorul DCL, ci de diferite utilitare apelate. Aceste linii nu încep cu „$".

În procedurile de comandă se pot introduce și comentarii, acestea fiind precedate de caracterul „!". De asemenea, se pot folosi etichete la instrucțiuni (linii), care sînt șiruri de caractere terminate prin „:".

În timpul tastării unei proceduri de comandă este indicat să se folosească comanda SET VERIFY, care va afișa pe măsură ce se execută fiecare comandă DCL, putîndu-se astfel detecta locul de producere a anumitor erori.

3.6.5. Apelarea procedurilor de comandă

Există două metode de operare a procedurilor de comenzi: interactiv și *batch*.

**Apelarea procedurilor de comandă în mod interactiv* se face primind caracterul „c" în fața numelui procedurii. Fișierul trebuie să fie de tip .COM. Apelul unei proceduri de comandă se poate face prin comanda DCL sau dintr-o altă procedură de comandă.

Să presupunem, de exemplu, că dorim apelarea procedurii din fișierul PROC.COM, în directorul implicit. Vom folosi comanda:

```
$ @SYS$LOGIN:PROC
```

**Apelarea procedurilor de comandă în mod batch.* Pentru execuția în paralel cu sesiunea (procesul) interactivă, se folosește comanda DCR SUBMIT. Ea are rolul de a plasa programul într-un fir de așteptare la execuție. Este indicată utilizarea unor proceduri în batch, atunci cînd este

vorba de programe ale căror execuții durează mult și ar bloca inutil terminalul de lucru:

```
$SUBMIT PROC
job proc (quene SYS$BATCH entry 210) started
```

Firul de așteptare SYS$BATCH este utilizat implicit dacă nu se folosește un alt fir specificat cu calificatorul /QUEUE.

3.6.6. Folosirea simbolurilor și funcțiilor lexicale. Proceduri de comandă

Simbolurile pot fi locale sau globale. Un *simbol local* este definit cu un singur semn egal (=) și este accesibil DCL-ului la nivelul comenzii la care a fost definit, precum și în nivelurile impuse în aceasta. De exemplu,

```
$CONTOR = 15*25
```

va defini un simbol local CONTOR. Orice simbol global este necunoscut doar pe durata procedurii în care a fost definit.

Un *simbol global* este definit cu 2 semne (= =). El va fi recunoscut la nivelul tuturor comenzilor DCL sau al procedurilor de comandă, pe timpul sesiunii de lucru; de exemplu,

```
$FIȘIER = = „@SYS$LOGIN:FIS"
```

În definirea numelui de simboluri se poate folosi și caracterul /*/, care indică locul de unde poate fi abreviat simbolul. De exemplu, simbolul FIS*IER poate fi apelat prin FIS, FISI, FISIE sau FIȘIER.

Funcțiile lexicale permit obținerea unor informații care pot fi vizualizate și cu comanda DCL SHOW. De exemplu, următoarele două comenzi sînt echivalente, ducînd la obținerea datei și orei:

```
$ SHOW TIME                          !comanda DCL
  25-JUN-1989 10:00:00
$ WRITE SYS$OUTPUT F$TIME()
                                     !funcție lexicală
  25-JUN-1989 10:00:00
$
```

În cadrul procedurilor de comandă se poate manipula ușor informația provenită din funcțiile lexicale. Fie următoarea procedură:

```
$!      EXEMPLU DATA.DAT
$!
$ TIME&DATE = F$TIME( )
$ DATE = F$EXTRACT (0, 11, TIME & DATE)
$ WRITE SYS$OUTPUT DATE
```

Se extrage din șirul de caractere, corespunzător datei și axei, doar data, care este apoi afișată. La apel se va obține:

```
$@DATA.DAT
25-jun-1989
$
```

Același efect se obține folosind o singură comandă:

```
$ WRITE SYS$LOGOUTPUT (F$EXTRACT) (0, 11,
F$TIME())
```

3.6.7. Intrări/ieșiri în proceduri de comandă

O posibilitate de a furniza unei proceduri anumite date de intrare constă în folosirea parametrilor. Există 8 astfel de simboluri rezervate (P1, ..., P8), care pot fi folosite în acest scop. Primului parametru îi este atribuit simbolul P1, celui de al doilea P2 ș.a.m.d. Valorile pentru parametri vor fi furnizate în momentul procedurii de comandă.

Spre exemplificare, fie procedura PROD.COM care va face înmulțirea a două numere întregi furnizate sub formă de parametri:

```
$!EXEMPLU PROD.COM
$!
$WRITE SYS$OUTPUT P1*P2
```

În cazul apelului procedurii, parametrii se dau pe linia de comandă:

```
$@PROD.COM 15 25
375
$
```

Modul cel mai uzual de introducere și extragere a datelor este însă cel interactiv. În acest scop, se pot utiliza următoarele comenzi: READ, WRITE, INQUIRE și TYPE.

**TYPE* este folosită pentru afișarea conținutului unui fișier pe terminal. Ea poate fi utilizată doar pentru afișarea unor șiruri text;

**WRITE* este oarecum similară cu TYPE, dar mai flexibilă. Ea este prelucrată de DCL, care va substitui în corpul comenzii toate simbolurile și funcțiile lexicale cu valorile acestora. Expresia care se va afișa va fi convertită la un șir de caractere;

**INQUIRE* permite acceptarea unor date de la tastatura terminalului. Se va afișa un prompter și se vor citi datele care se vor asigna simbolului precizat. Toate datele sînt acceptate ca șiruri de caractere. Se face conversia tuturor caracterelor în caractere mari;

**READ* permite citirea unor date de pe un periferic specificat (care poate fi tastatura). Se afișează prompterul după care se citesc datele și se afectează unor simboluri indicate, fără a se face nici o conversie.

3.6.8. Utilizarea fișierelor în proceduri de comandă

În vederea scrierii/citirii unor date de pe /de pe fișiere se pot folosi comenzile: OPEN, CLUSE, READ și WRITE.

**Comanda OPEN* permite deschiderea unui fișier în vederea scrierii sau citirii sale. Deschiderea se poate face folosind următorii calificatori:

CALIFICATORI:

/READ	pentru deschidere la citire
/WRITE	pentru deschidere la scriere, cu crearea unui nou fișier
/APPEND	pentru deschidere la scriere în continuare pe un fișier existent

**Comanda CLOSE* permite închiderea fișierului indicat.

3.6.9. Instrucțiuni pentru controlul secvenței de execuție

Procedurile de comandă nu sînt formate numai din comenzi DCL, care se execută secvențial. Ele pot conține și o serie de instrucțiuni ce permit realizarea unor structuri de programe mai complexe, care să conțină pe lîngă secvență, decizia și ciclurile.

Instrucțiunea IF permite codificarea deciziilor, adică a unor condiții. Sintaxa ei este următoarea:

```
$IF condiție THEN comanda
```

Se evaluează *condiția*, al cărei rezultat este boolean. Dacă rezultatul este adevărat se execută *comanda*, altfel se trece la instrucțiunea următoare. Iată în continuare un exemplu de utilizare a instrucțiunilor:

```
$ ! EXEMPLU TIMP.COM
$ !
$TIMP = F$TIME( )                    !obține timpul
$ORA=F$EXTRACT (12, 2, TIMP)
                                     !extrage ora
$IF ORA. LT. 10 THEN
$  WRITE SYS$OUTPUT „Bună dimineața"
$EXIT
```

Instrucțiunea GOTO permite realizarea unor salturi necondiționate. Sintaxa ei este:

```
$ GOTO eticheta
```

Iată același exemplu de mai sus, modificat astfel încît să se folosească instrucțiunea GOTO:

```
$ ! EXEMPLU TIMP.COM
$ !
$ TIMP=F$TIME( )
$ ORA=F$EXTRACT (12,(2, TIMP)
$ IF ORA.LT.10 THEN GOTO DIMINEAȚA
$ IF ORA.LE.18 THEN GOTO ZIUA
$ IF ORA.GT.18. THEN GOTO SEARA
$ OTO END
$ DIMINEAȚA:
$     WRITE SYS$OUTPUT „BUNĂ DIMINEAȚA"
$     GOTO END
$ ZIUA:
$     WRITE SYS$OUTPUT „BUNĂ ZIUA"
$     GOTO END
$ SEARA:
$     WRITE SYS$OUTPUT „BUNĂ SEARA"
$     GOTO END
$ END:
$     EXIT
```


Codificarea ciclurilor se poate face în două moduri:

— *cu test inițial*, folosind următoarea schemă:

```
$LOOP:
$     IF .NOT. condiție THEN GOTO END
          .
          .
          . instrucțiunile ciclului
$     GOTO LOOP
$ END:
$     EXIT
```

— *cu test final*, utilizînd schema:

```
$ LOOP
          .
          .
          . instrucțiunile ciclului
          .
$ IF condiție THEN GOTO LOOP
$ EXIT
```

Subrutinele se pot folosi utilizînd comenzile GOSUB, care transferă controlul la o anumită etichetă ce reprezintă începutul subrutinei și RETURN care termină subrutira și face revenirea la instrucțiunea de după GOSUB. Utilizarea subrutinelor face ca programele să fie modulare și ușor de citit.

În exemplul următor schițăm utilizarea subrutinelor în proceduri de comandă:

```
$ ! EXEMPLU FIȘIER.COM
$ !
$ ! deschide fișier
$ OPEN/WRITE FIS FISPERS.DAT
$ ! colectează date-
$ INQUIRE ARTICOL „numele"
$ GOSUB SCRIE-FIȘIER
$ INQUIRE ARTICOL „adresa"
```

```
$ GOSUB SCRIE-FIŞIER
$ CLOSE FIS
$ EXIT
$!
$! subrutina SCRIE-FIŞIER
$ SCRIE-FIŞIER:
$     WRITE FIS ARTICOL
$     RETURN
```

**Terminarea* procedurile de comandă se poate face cu:

EXIT, care termină procedura curentă şi returnează controlul nivelului apelant, ce poate fi o altă procedură de comanda sau DCL;

STOP, care termină toate procedurile de comandă, dînd controlul DCL-ului.

3.6.10. Exemple de proceduri de comandă

1) *Afişarea unor texte pe terminal*

```
$! introduceţi numele de la tastatură
$ NQUIRE NUME „numele"
$  WRITE SYS$OUTPUT „Salut", NUME
$! introduceţi numele fişierului
$  INQUIRE NUME-FIS „numele fişierului"
$! Citire valori de la tastatură
$  READ/PROMT=„Val1=" SYS$INPUT VAL1
$  READ/PROMT=„Val2=" SYS$INPUT VAL2
$! scrie la terminal valorile şi sumele lor
$  WRITE SYS$OUTPUT VAL1, „+", VAL2, „=",
     VAL1+VAL2
$! Afişează fişierul indicat NUME-FIS
$  TYPE NUME-FIS
$! Afişează director implicit
$  WRITE SYS$OUTPUT F$DIRECTORY()
$  EXIT
```

2) *Scriere pe fişiere disc*

```
$! deschide fişiere
$!
$  OPEN/WRITE FIS FIŞIER.DAT
$!
```



```
$! Scriu în fişier data şi ora
$!
$  WRITE FIS „Acest articol s-a scris la:„,F$TIME()
$! Închid fişier
$  CLOSE FIS
$  EXIT
```

3) *Actualizare fişiere*

```
$! Deschide fişier
$!
$  OPEN/APPEND FIS FIŞIER.DAT
$! scriem la sfîrşit
$  WRITE FIS „Acel articol s-a închis la „,F$TIME()
$! Închidem fişierul
$  CLOSE FIS
$  EXIT
```

4) *Citire fişier*

```
$! Deschide fişier
$!
$  OPEN/READ FIS FIŞIER.DAT
$! Citire articol
$  READ FIS ARTICOL
$  WRITE SYS$OUTPUT „Primul articol !„,ARTICOL
$! Închidem fişier
$  CLOSE FIS
$  EXIT
```

5) *Procedura de iniţializare a sesiunii de lucru*

```
$! EXEMPLU LOGIN.COM
$!
$! Se execută la intrarea în sesiune
$!
$! Verifică dacă conectarea este în reţea sau batch
$!
$  IF F$MODE ( ) .EQS. „NETWORK" THEN GOTO
     RETEA
$  IF F$MODE( ) .EQS. „BATCH" THEN GOTO
     BATCH
$!
```

```
$ ! Definirea unor simboluri permanente
$ !
$  SD==„SET DEFAULT“
$  SH==„SET HOST“
$  WI*DE==„SET TERMINAL/WIDTH=132“
$  NA*RROW==„SET TERMINAL/WIDTH=80“
$  DIRRECTORY==„DIRECTORY/SIZE“
$  PURRGE==„PURGE/LOG/KEEP=2“
$  HOHME==„SET DEFAULT SYS$LOGIN:“
$  WHO==„SHOW USERS“
$  EVE==„EDIT/TPU“
$  EDT==„EDIT/EDT/COMMAND=SYS$LOGIN:
     EDTINI.EDT“
$  BR*OWSE==„TYPE/PAGE“
$ !
$ ! DEFINESC CHEI SPECIALE
$ !
$  DEFINE/KEY/NOLOG/TERM PF1 „DIR“
$  DEFINE/KEY/NOLOG  PF2 „EDIT“
$  DEFINE/KEY/NOLOG/TERM/NOECHO PF3
     „LOGOUT“
$  DEFINE/KEY/NOLOG/TERM/NOECHO HELP
     „SHOW KEY/ALL“
$ !
$ ! Modific caracteristicile terminalelor
$ !
$  SET TERMINAL/INSERT
$  SET PROMPT =„DAN >“
$ !
$ ! Afișează data și ora
$ !
$  SHOW TIME
$ !
$ ! Conectare în rețea
$ !
```



```
$  NETWORK:
   . comenzi necesare la conectare în retea
$  EXIT
$ !
$ ! Conectare batch
$ !
$  SET VERIFY
$  EXIT
```

6) *Căutare subdirector*

```
$ ! EXEMPLU SUBDIR.COM
$  START:
$  WRITE SYS$OUTPUT F$DIRECTORY ()+
     + „SUBDIRECTOARE:“
$  WRITE SYS$OUTPUT“ “
$ !
$ ! Ciclul de căutare subdirector și afișare a numelui lor
$ !
$  DIR$LOOP:
$  FIS=F$SEARCH („*DIR“)
$ !
$ ! Dacă DCL returnează șirul nul, s-a terminat
$ !
$  IF FIS.EQS. „ “ THEN GOTO END
$ !
$ ! Găsește poziția punctului în numele fișierului
$ !
$  PUNCT=F$LOCATE („.“, FIS)
$ !
$ ! Găsește poziția parametri stînga
$ !
$  PAR=F$LOCATE („“, FIS)
$ !
$ ! Extrage șirul dintre punct și paranteză (numele)
$ !
$  FIS=F$EXTRACT (PAR+1, PUNCT-PAR-1, FIS)
$ !
```

```
$ ! Afişează numele subdirectorului
$ !
$ WRITE SYS$OUTPUT „... 'FIS' "
$ GOTO DIR$LOOP
$ END:
$ EXIT.
```

3.7. PROGRAME UTILITARE

3.7.1. Utilitarul HELP

Utilitarul HELP reprezintă una din cele mai importante facilităţi disponibile în VAX/VMS. El pune la dispoziţia utilizatorilor echivalentul unui dicţionar cu comenzile DCL. Se furnizează, la ieşire, explicaţii pentru fiecare comandă DCL şi, opţional, pentru fiecare parametru (calificator) aferent.

HELP poate fi folosit în 2 moduri. Cînd se ştie comanda DCL, dar se doresc mai multe informaţii despre ea, se foloseşte *în mod direct*. Dacă nu se ştie numele comenzii, se utilizează *modul interogativ*.

a) *HELP — în modul direct*

În acest mod de lucru se pot obţine informaţii despre o comandă al cărei nume se cunoaşte. Se va tasta numele unei comenzi DCL cu sau fără calificator. De exemplu, dacă vrem informaţii despre comanda DCL SET cu calificatorul /TERMINAL, vom tasta:

$HELP SET TERMINAL

b) *HELP — în modul interogativ*

În acest mod se tastează HELP ⟨RETURN ⟩; ca urmare se vor afişa toate comenzile DCL, în ordine alfabetică. Apoi utilizatorul este întrebat care comandă doreşte să fie detaliată:

Topic"

După ce s-a tastat numele unei comenzi, HELP afişează informaţiile referitoare la ea, inclusiv lista calificatorilor. Apoi este întrebat dacă doreşte informaţii despre unul din calificatori:

NUME-COMANDA Subtopic":

Ieşirea din HELP se face cu CTRL/Z.

Pentru mai multe informaţii despre utilizatorul HELP, se pot utiliza comenzile:

$HELP HINTS

$HELP HELP/INSTRUCTIONS

7.7.2. Utilitarul BACKUP

BACKUP este un utilitar necesar inginerului de sistem şi operatorilor pentru salvarea/restaurarea informaţiilor de pe/pe discuri.

În general, aceste operaţii se execută pe bandă magnetică, ce reprezintă cel mai folosit suport de arhivare. Dar BACKUP poate face transferuri între discuri. Qperaţiile de salvare/restaurare se referă la anumite fişiere sau la toate fişierele sistemului. În acest din urmă caz, utilizatorul trebuie să dispună de privilegii speciale. Utilizatorii obişnuiţi folosesc BACKUP pentru salvarea fişierelor proprii.

În momentul apelului lui BACKUP, prin intermediul calificatorilor se poate selecta unul din următoarele *moduri de lucru*:

SELECTIV, cînd BACKUP va salva/restaura fişiere după criterii ca data, tipul fişierului, versiune. De exemplu, calificatorul /DATE va selecta fişierele după date;

FIŞIER CU FIŞIER, cînd salvarea/restaurarea se face copiind fişiere sau directoare întregi, indicate prin numele lor.

INNCREMENTAL, cînd se salvează fişiere create după data ultimei operaţii BACKUP.

FIZIC, cînd se salvează/restaurează întregul volum. Copierea se face bit după bit, ignorîndu-se structura fişierelor.

IMAGINE, cînd se creează o copie funcţională, echivalentă, a volumului original.

COPIERI DISC-DISC

BACKUP permite execuţia unor copieri de volum, directoare şi fişiere, de pe un disc pe altul sau pe acelaşi disc.

a) Pentru a *copia fișiere*, comanda are forma:

$BACKUP fișier1 fișier2

Argumentul fișier1 este specificatorul fișierului sursă, iar fișier2, al fișierului destinație. Dacă fișierul destinație se specifică clar sub forma unui director, BACKUP va păstra numele fișierului sursă. De exemplu, pentru a copia fișierul LOGIN.COM în directorul [CATEDRA.GRUPA], se va folosi comanda:

$BACKUP LOGIN. COM [CATEDRA.GRUPA]

sau comanda:

BACKUP LOGIN.COM [CATEDRA.GRUPA] LOGIN. COM

Pentru copierea mai multor fișiere, putem tasta de exemplu:

$BACKUP*.PAS [CATEDRA.GRUPA]

b) Pentru a *copia directoare*, se folosește o comandă cu următorul format:

$*BACKUP* [*director...*] [*fișier*] [*director...*] *fișier*

Se copiază toate directoarele și fișierele din specificatorul sursei în cel al destinației. Dacă acesta nu există pe disc, se creează un astfel de director.

De exemplu, comanda:

$*BACKUP* [*GRUPA... *]. *FOR DUA2*: [*PROGRAME FORTRAN*]

va determina copierea tuturor fișierelor de tip. FOR din directorul GRUPA în directorul FORTRAN, de pe discul DUA2:

c) Pentru *copierea de volume*, se face mai întîi montarea volumului destinație, apoi copierea cu calificatorul /IMAGE:

$MOUNT/FOREIGN DUA2:

$BACKUP/IMAGE DUA1:.DUA2:

Copieri disc-banda

Pentru a putea lucra prima dată cu un volum bandă, el trebuie să fie inițializat:

$INIȚIALIZE MUAO:BANDA

Inițializarea are ca efect ștergerea tuturor fișierelor dacă banda a mai fost utilizată

a) *Copierea de fișiere pe bandă*

Se face montînd mai întîi volumul bandă și apoi lansînd comanda de copiere, ca în exemplul:

$MOUNT/FOREIGN MUA 0:

$BACKUP [STUDENTI] P1.PAS MUAO: [P1.PAS]

b) *Salvarea fișierelor disc pe banda*

Se poate face individual sau un director întreg. Pentru *salvarea unui fișier* procedăm astfel:

$MOUNT/FOREIGN MUAO:

BACKUP DUB1: [STUDENȚI] *.*;* MUAO:FIȘIER. BAK/SAVE-SET

Comanda salvează toate fișierele din directorul STUDENȚI pe banda montată pe MUAO: într-un set de volum numit FIȘIER. BAK.

Pentru *salvarea unui director întreg* se procedează astfel:

$MOUNT/FOREIGN MUAO:

$BACKUP DUB1: [STUDENȚI...] MUAO:FIȘIERE. BAK/SAVE-SET

Comanda va salva toate fișierele din directorul STUDENȚI și din subdirectoarele sale. Pentru a salva doar fișierele din directorul STUDENȚI nu se mai pun ... .

c) *Restaurarea fișierului disc de pe bandă*

Pentru a *restaura fișiere*, procedăm astfel:

$MOUNT/FOREIGN MUAO:

$BACKUP

— FROM:MUAO:FIȘIERE.BAK/SAVE-SET/SELECT = [STUDENȚI] LOGIN.COM

— TO:*.*

Această comandă va restaura fișierul LOGIN.COM în directorul implicit. Calificatorul /SELECT permite selectarea unui anumit fișier din setul de salvare indicat (FIȘIERE.BAK).

Pentru a *restaura un director* întreg procedăm astfel:

$MOUNT/FOREIGN MUAO:

$BACKUP MUAO:FIȘIERE.BAK/SAVE-SET DUB1: [STUDENTI...]

Comanda va restaura toate fișierele din setul de salvare în directorul rădăcină STUDENȚI și în subdirectoarele sale.

d) *Listarea unui set de salvări*

Pentru a lista conținutul unui set de salvare de pe bandă, creat cu BACKUP, se va folosi opțiunea /LIST:

```
$MOUNT/FOREIGN MUAO:
$BACKUP/LIST MUAO:FIȘIERE.BAK/SAVE-SET
```

Informațiile listate includ directorul, numele fișierului, tipul, versiunea, dimensiunea în blocuri, data creării pentru fiecare fișier salvat în altul.

Alte facilități BACKUP

Utilizatorul BACKUP recunoaște și o serie de alți calificatori, dintre care amintim cîțiva mai importanți:

/SINCE — care permite salvarea/restaurarea selectivă, în funcție de data creării fișierului;

/VERIFY — care determină modul în care s-a făcut copierea;

/CONFIRM — afișează numele fiecărui fișier și așteaptă confirmarea de salvare/restaurare;

/DELETE — șterge fișierele sursă după copiere.

3.7.3. Utilitarul LIBRARY

LIBRARY este un utilitar de tip bibliotecar, ajutînd utilizatorii să creeze și să întrețină un fișier cu format special (biblioteca), în care se memorează grupuri de fișiere numite *module*. Bibliotecile predefinite includ fișiere text, help, obiect, imagine și macro. Aici utilizatorii își pot depune programe sursă, documente și texte, programe executabile sau subprograme.

Crearea bibliotecilor se face apelînd programul LIBRARY cu doi calificatori, /CREATE și un al doilea care indică tipul și poate fi:

```
/TEXT
/SHARE
/HELP
/OBJECT
/MACRO
```

De exemplu, comanda:

```
$LIBRARY/TEXT/CREATE CĂRȚI
```

va crea o bibliotecă numită CĂRȚI, care va conține modele de tip TEXT.

La crearea unei biblioteci se poate indica și lista fișierelor ce vor fi incluse în ea:

```
$LIBRARY/TEXT/CREATE CĂRȚI F1, DOC, FIS,
INDEX
```

LISTAREA CONȚINUTULUI UNEI BIBLIOTECI se face utilizînd calificatorul /LIST, ca în următorul exemplu:

```
$LIBRARY/TEXT/LIST CĂRȚI
```

Apoi utilitarul afișează lista modulelor din bibliotecă.

Adăugarea de noi module în bibliotecă se face utilizînd calificatorul /INSERT:

```
$LIBRARY/TEXT/INSERT CĂRȚI CAP1, CAP2
```

Extragerea unui modul dintr-o bibliotecă sub forma unui fișier într-un director, se face cu calificatorul /EXTRACT:

```
$LIBRARY/TEXT/EXTRACT CĂRȚI CAP1
```

Inlocuirea unui modul din bibliotecă cu un alt fișier se face cu calificatorul /REPLACE:

```
$LIBRARY/TEXT/REPLACE CĂRȚI CAP2
```

3.7.4. Utilitarul SORT

SORT permite reorganizarea articolelor unui fișier, realizînd sortarea lor după diferite criterii. În cazul cel mai simplu aceasta se face în ordine alfabetică, opțiunea fiind implicită. De exemplu, vrem să ordonăm alfabetic articolele din fișierul PERS.TXT; rezultatul sortării depunîndu-l în fișierul SORT-PERS.TXT. Pentru a realiza acest lucru folosim comanda:

```
$SORT PERS.TXT SORT-PERS.TXT
```

Există însă și situații cînd sortarea trebuie făcută după un anumit cîmp de articole — alfabetică sau numerică, ascendentă sau descendentă. Pentru aceasta folosim calificatorul /KEY, ca în exemplul următor:

```
$SORT/KEY=(POSITION=10, SIZE=4, DESCEND-
ING) F1.TXT F2.TXT
```

Comanda va sorta fișierul F1.TXT, descrescător, conform unei chei formate din 4 caractere și care începe în poziția 10. Rezultatul sortării se depune în fișierul F2.TXT.

3.7.5. Utilitarul MAIL

MAIL este un utilitar care permite emisia sau recepția de mesaje de poștă electronică, memorîndu-le în sistem. Utilizatorii pot comunica între ei prin poștă electronică, utilitarul MAIL punînd la dispoziție facilități de emisie, citire, trimitere mai departe sau răspuns la mesaje de tip scrisoare.

Dacă se emite o scrisoare către un alt utilizator, acesta este atenționat prin apariția unui mesaj pe ecran. Mesajul îi indică că a sosit poșta pe adresa lui pe care va putea să o consulte.

Dacă destinatarul mesajului nu este cuplat, mesajul de atenționare îi va apărea în momentul inițializării de către acesta a sesiunii de lucru.

MAIL este un utilitar interactiv, apelat prin tastarea comenzii:

```
$MAIL
```

Emisia de scrisori (mesaje) se face folosind, după apelul lui MAIL, comanda SEND (emite). Sîntem întrebați asupra destinatarului, a subiectului scrisorii, după care va trebui să tastăm mesajul, pe care îl încheiem cu ⟨CTRL/Z⟩ (EOF):

```
$MAIL
MAIL >SEND
TO:DAN
SUBJ: ORE
Enter your message:
   Profesore,
Mîine la ora 10 ai laborator la PROGRAMARE
      Cu bine,
         VAL ⟨CTRL/Z⟩
```

Mesajele pot fi trimise și la mai mulți utilizatori simultan. De asemenea, utilizînd comanda SEND/EDIT se apelează editorul EDT care permite o elaborare mai ușoară a textului scrisorii.

Emisia unui fișier drept scrisoare se poate face direct în linia de comandă DCL:

```
$MAIL fișier utilizator/subject="sir"
```

unde *fișier* este specificatorul fișierului ce va fi emis, iar *utilizator* este numele destinatarului.

Utilizarea listelor de distribuție

În mod frecvent, mesajele se trimit la același grup de utilizatori; pentru a economisi timp, se poate crea un fișier numit *listă de distribuție*. În el se scrie numele fiecărui utilizator care va recepționa mesajul. Cînd se dorește limitarea mesajului, la întrebarea „TO:" se va tasta *@FIȘIER*, unde *fișier* este specificatorul lista de distribuție.

CITIREA UNEI SCRISORI se face în felul următor:

```
$MAIL
```

YOU HAVE ... MESSAGES

Se va tasta ⟨RETURN⟩ și se vor afișa toate scrisorile pe ecran. Utilizatorul poate răspunde la un mesaj, prin utilizarea comenzilor REPLAY sau ANSWER.

În cazul în care se dorește citirea unei anumite scrisori se folosește comanda READ:

```
   $MAIL
MAIL >READ 10
$10          25-09-1990      20:00:00      MAIL
FROM: VAL
  TO  : DAN
SUBJ: ORE
   |  /
   |  /
         scrisoare
   |  /
   |  /
```

Fișiere „cutii poștale"

Sistemul utilizează în mod implicit trei fișiere „cutie poștală":

NEWMAIL — pentru memorarea scrisorilor emise, dar care nu au fost citite;

MAIL — pentru memorarea scrisorilor după ce au fost citite. Ele trec automat din NEWMAIL în MAIL;

WASTEBASKET — pentru memorarea scrisorilor șterse (cu comanda DELETE), care sînt reținute pînă la ieșirea utilizatorului MAIL. Ele pot fi recuperate din acest fișier cu comanda MOVE, în cazul în care ne-am răzgîndit cu privire la ștergerea unei scrisori.

Utilizatorul își *poate cere propriile sale fișiere „cutie poștală"*. Apoi, cu comanda MOVE, poate muta o scrisoare (selectată prin numărul ei) din cutiile sistem în acest fișier. Iată un exemplu:

```
$MAIL
MAIL >10
MAIL >MOVE ARHIVA
Folder ARHIVA does not exist. Do you want create
it(y/n):Y
MAIL >
```

Ștergerea unei scrisori se face cu comanda DELETE:

```
$MAIL
MAIL >DELETE 10
```

În cazul în care se dorește ștergerea tuturor scrisorilor dintr-o cutie poștală, se procedează astfel:

```
$MAIL
MAIL >SELECT ARHIVA ! selectez „cutia"
MAIL >DELETE/ALL
```

3.7.6. Utilitarul PHONE

PHONE permite să se poarte „convorbiri" între utilizatorii conectați la sistem. Utilizatorul simulează funcționarea și facilitățile sistemului telefonic real.

Apelul unui utilizator se face cu comanda:

```
$PHONE utilizator
```

În acest moment ecranul se împarte în două zone orizontale. În zona superioară apar mesajele trimise de utilizator, iar în cea inferioară cele sosite ca răspuns de la partener.

Apelul lui PHONE se poate face și interactiv:

```
$PHONE
```

La apariția prompterului (%) se pot introduce comenzi, printre care: DIRECTORY, care duce la afișarea utilizatorilor conectați în acel moment, HELP, care permite afișarea facilităților PHONE.

Răspunsul la apel se face cu comanda ANSWER:

```
$PHONE
%ANSWER
```

În acest moment dialogul poate începe în mod efectiv.

3.7.7. Alte utilitare disponibile sub VMS

Există un număr mare de alte utilitare care pot fi folosite sub sistemul de operare VMS, dintre care amintim:

ACMS (*A*pplication *C*ontrol and *M*anagement *S*ystem), un sistem destinat dezvoltării de aplicații multiutilizator: proiectarea datelor, proiectarea meniurilor (cu TDMS) și proiectarea taskurilor (cu utilitarul ADL)

FMS, un ansamblu de programe pentru construirea/ /gestiunea de videoformate, având mai multe componente:

— editorul de videoformat;
— bibliotecar pentru catalogarea videoformatelor;
— tester de videoformate;
— set de funcții aplicabile din limbaj de asamblare și limbaje de nivel înalt;
— utilitar de tip „HELP".

TDMS (*T*erminal *D*ata *M*anagement *S*ystem), un grup de utilitare destinat realizării de aplicații interactive care folosesc videoformate. Se permite astfel utilizatorilor să definească, în afara programelor de aplicație, funcții de I/E de la terminale. În realizarea aplicațiilor trebuie parcurse mai multe etape:

— definirea machetelor de I/E (cu utilitarul FDU);
— definirea structurii fișierelor (cu CDDL sau DTR ori sisteme de baze de date DBMS);
— definirea cererilor de I/E (cu RDU);
— crearea de biblioteci de cereri de I/E la terminale;
— scrierea programelor în BASIC, COBOL, FORTRAN, care pe baza rutinelor TDMS să exploateze bibliotecile de cereri.

ALLN1, un software de birou, folosit ca agendă de lucru, editare, gestiune documente, calculator de birou, comunicații cu alți utilizatori, telefon electronic, poștă electronică, schimb de mesaje. Pachetul de programe a fost realizat utilizînd FMS; ca urmare, există biblioteci de videoformate care sînt accesibile utilizatorilor și permit schimbarea limbii de dialog. Dintre tipurile de aplicații enumerăm:

— ***agenda de lucru*** — permite evidența activităților de planificare ale utilizatorului: ședințe, întîlniri de lucru,

activități urgente (afișate ordonat la începutul sesiunii). Se poate avea acces la agendele de lucru ale partenerilor;

— *agenda de telefoane și adrese*, se creează în instituții, la ea avînd acces toți salariații. Există posibilitatea căutării în agendă după diferite criterii;

— *editare/gestiune documente*, permite fiecărui utilizator să aibă o bibliotecă de documente sau acces la grupul comun de documente. Într-o „cartotecă" se păstrează dosarul, titlul, numărul de catalogare. Pentru construirea documentelor se poate folosi EDT, dar și un alt editor performant (WPS). Fiecărui document i se pot asocia mai multe cuvinte cheie;

— *calculator de birou*, permite afișarea pe ecran a unui calculator în vederea efectuării unor operații aritmetice;

— *poștă electronică*, permite comunicarea între utilizatori prin intermediul „scrisorilor". Fiecare utilizator dispune de propriul său fișier de „cutie poștală".

— *telefonul electronic*, permite folosirea ecranului pe post de telefon pentru emisia/recepția directă de mesaje între utilizatori.

Sisteme de gestiune de baze de date:

— DATATRIVE — regăsire de date în fișiere;

— DBMS-11 —SGBD de tip CODASYL al firmei DEC;

— RDB — SGBD relațional pentru VAX;

— TEAMDATA — facilități interactive de regăsire (cu meniuri) a informațiilor din baze de date construite cu sistemele anterioare.

— ORACLE — sistem de baze de date relaționale.

VAX-set constituie un ansamblu de programe pentru dezvoltarea de aplicații informatice și pentru întreținerea lor. Se furnizează un ansamblu de „unelte" software pentru elaborarea produselor program:

LSEDIT (*L*anguage *S*enzitive *E*ditor);
SMC (*C*ode *M*anagement *S*ystem);
MMS (*M*odule *M*anagement *S*ystem);
DTM (*D*ata *T*est *M*anager);
SCA (*S*ourse *C*ode *A*nalyzer);
PCA (*P*erformance and *C*onversage *A*nalyzer);
DBG (*S*ymbolic *DeBuGger*).

Capitolul 4

SISTEMUL DE OPERARE MS - DOS

Sistemul de operare MS-DOS este destinat familiei de calculatoare personale compatibile IBM-PC, realizate în jurul unuia dintre microprocesoarele 8086, 80186, 80286, sau 80386. În țara noastră a fost realizat sistemul de operare DOS-PC, compatibil cu MS-DOS, care funcționează pe calculatoare de fabricație românească M-216, Felix-PC, Junior-XT etc.

MS-DOS este un sistem de operare monoutilizator; el poate lucra multitasking numai pentru configurații cu 80286 sau 80386. Sistemul prezintă o interfață simplă, ușor accesibilă, care permite o bună gestiune a resurselor hardware și software.

Printre principalele facilități oferite de MS-DOS putem aminti:

— copierea, compararea, afișarea, ștergerea și redenumirea fișierelor;

— formatarea discului;

— execuția programelor sistem sau utilizator;

— selectarea opțiunilor pentru ecran și imprimantă;

— transferul sistemului de operare pe un alt disc;

— tipărirea unui fișier la imprimantă în timp ce sistemul de operare execută alte comenzi;

— sortarea sau căutarea de șabloane în fișiere text;

— schimbarea prompter-ului;

— conversia unui fișier executabil (.EXE) în programe compatibile cu cele .COM.

Una din calitățile majore ale sistemului MS-DOS este diversitatea foarte mare și calitatea foarte bună a resurselor software:

— compilatoare și medii de programare (Turbo PASCAL, Turbo C, Turbo PROLOG, MODULA-2, ADA, FORTRAN-77, C ++);
— interpretoare (BASIC, QUIK BASIC);
— asambloare (ASM, MASM);
— editor de texte (EDLINK, WORDSTAR, EDIT);
— sisteme pentru gestiunea bazelor de date (DBASE-II, DBASE-III, DBASE-III PLUS, DBASE-IV-PLUS);
— generatoare de rapoarte (MULTIPLAN, LOTUS 1-2-3);
— programe pentru lucrul în rețea (KERMIT, NETWARE);
— utilitare pentru gestiunea sistemelor de fișiere (XTREE, PC TOOLS, WINDOWS), PC SHELL, DOS SHELL);
— un mare număr de aplicații specifice diverselor domenii de utilizare (grafică, proiectare asistată, gestiune).

4.1. STRUCTURA SISTEMULUI MS-DOS

Sistemul de operare MS-DOS se compune din următoarele elemente:

— BIOS (*B*asic *I*nput *O*utput *S*istem), rezident in PROM;
— artiocolul cu programul încărcător al sistemului de operare (*bootstrap*);
— programul IBMBIO.COM;
— programul IBMDOS.COM;
— programul COMMAND.COM;
— programe utilitare: FORMAT, DISCKOPY, CHKDISK etc.

În fig. 4.1 este ilustrată legătura dintre componentele sistemului MS-DOS.

• *BIOS* reprezintă sistemul de programe de intrare/ieșire de bază. El este rezident în PROM și conține următoarele elemente:

— programe de test pentru toate resursele din configurația standard a sistemului; se face la inițializarea sistemului și vizează microprocesorul, memoria RMA, sistemul de întreruperi, adaptare pentru periferice etc.;

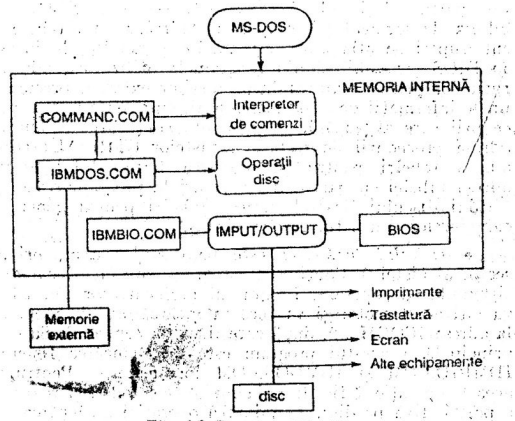


Fig. 4.1. Structura MS—DOS

— programul prefncărcător pentru încărcarea articolului cu încărcătorul sistemului de operare (*bootstrap* — sectorul 1 al pistei 0 a discului sistem);

— proceduri pentru tratarea întreruperilor externe (de la ceasul de timp real, de la tastatură și de la discul flexibil);

— procedurile pentru comanda și controlul execuției operațiilor de intrare/ieșire la nivel fizic pentru toate perifericele din configurația standard.

Programele utilizator pot avea acces la procedurile conținute în BIOS, pe bază unei întreruperi software (instrucțiunea INT). În memoria RAM, începînd cu adresa 000 : 000, se găsește tabela care conține adresele rutinelor de tratare a întreruperilor. Ea este alcătuită din intrări de

cîte patru octeți: adresa relativă în segment (2 octeți) și adresa de segment (2 octeți). În cadrul acestei tabele de întreruperi se află adresele rutinelor punctelor de intrare în BIOS (accesibil și din programele utilizator), adresele rutinelor pentru tratarea întreruperilor externe, precum și unele informații necesare sistemului, cum sînt: adresa procedurii care afișează la imprimantă conținutul ecranului, adresa procedurii de tratare a tastelor CTRL/ALT/DEL, adresa tabelei pentru programarea terminalului grafic, adresa tabelei cu parametrii necesari la programarea adaptorului discului flexibil, adresa tabelei pentru generarea caracterelor din setul ASCII extins.

• *Articolul încărcător* este pe discul sistem, primul sector al pistei 0. Conține un program care face încărcarea sistemului de operare. Programul preîncărcător din BIOS va citi acest sector și va încărca încărcătorul în memorie la adresa 7COOH : 0, după care îi va da controlul. Sarcina principală a acestui program este de a încărca fișierele IDMBIO.COM și IBMDOS.COM în memorie. Pentru că această operație să fie cît mai simplă, cele două fișiere ocupă o poziție fixă pe disc. Terminarea operației de încărcare a sistemului de operare se face cu lansarea în execuție a programului COMMAND.COM.

• *Programul IBMBIO.COM* conține unele extensii ale BIOS-ului. El are rolul de a realiza actualizări ale programului BIOS corespunzătoare noilor versiuni de sistem de operare. În plus, acest program permite extinderea configurației standard prin adăugarea de noi echipamente periferice fără a fi necesară modificarea BIOS-ului din PROM. Utilizatorii pot adăuga noi drivere pentru controlul noilor echipamente.

• *Programul IBMDOS.COM* conține procedurile de tratare a operațiilor de intrare/ieșire la nivel logic cu discul. Apelul acestei proceduri DOS din programele utilizator se face folosind de asemenea instrucțiunea INT. Conform convențiilor Microsoft, există două tipuri de apeluri DOS: întreruperi și funcții.

• *Programul COMMAND.COM* este interpretorul de comenzi. El este format din patru componente:

— o componentă care este rezidentă în memoria internă, imediat după IBMDOS. COM. Această porțiunea tratează întreruperile de terminare program, CTRL/BREAK, precum și erorile de citire. De asemenea, această componentă la terminarea unui program calculează suma de control pe baza căreia se verifică dacă zona tranzientă a fost modificată. În caz afirmativ se reîncarcă componenta tranzientă;

— o componentă care realizează inițializarea sistemului de operare și care se execută la lansarea sistemului de operare. Această componentă verifică dacă există pe discul sistem un fișier cu numele AUTOEXEC.BAT. În caz afirmativ acesta se va executa, el constituind un fișier de comenzi indirecte. Zona de memorie ocupată de această componentă va fi acoperită de primul program care va fi încărcat de COMMAND.COM; din acest moment ea nu mai este necesară;

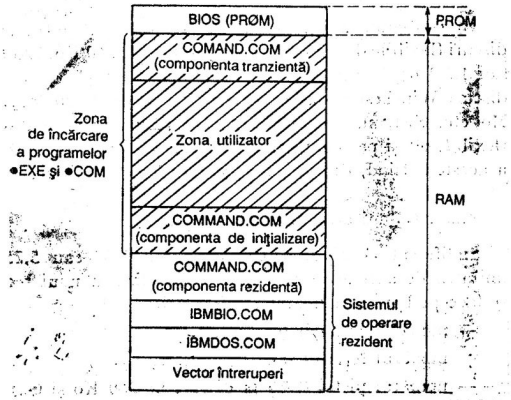


Fig. 4.2. **Componentele sistemului MS—DOS.**

— o componentă tranzitorie care va fi încărcată la sfîrșitul memoriei și conține comenzile interne (rezidente) ale MS-DOS (DIR, TYPE, COPY, DATA, TIME etc.), interpretorul de comenzi și de fișiere de comenzi indirecte;

— o componentă care afișează prompter-ul (de obicei, „C >"), citește comenzile de la tastatură sau de la fișierul de comenzi indirecte. Această componentă conține încărcătorul pentru programul utilizator sau sistemul cu extensiile .EXE (pentru programe executabile) și .COM (pentru fișiere imagine de memorie). La sfîrșitul execuției fiecărui program se verifică dacă partea din COMMAND.COM care interpretează linia de comandă mai este rezidentă în memorie. Dacă este necesar se face reîncărcarea acestui fișier.

În figura 4.2 sînt prezentate organizarea memoriei interne și amplasarea componentelor sistemului de operare.

4.2. ORGANIZAREA DISCULUI LA MS-DOS

În configurația gestionată în MS-DOS pot intra atît discuri flexibile de 5,25 inch cu capacitate de 360 Ko, 720 Ko sau 1,2 Mo; de 3,5 inch, cu capacitatea de 1,44 Mo, cît și discuri Winchester fixe cu capacități tipice de 20 Mo, 40 Mo etc. Sistemul de operare poate rezida atît pe discul flexibil, cît și pe cele fixe, structura și modul de organizare a acestora fiind, din punct de vedere logic, similare.

4.2.1. Organizarea discului flexibil

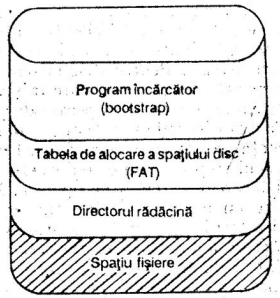
Indiferent de tipul discului flexibil (3,5" sau 5,25"), unitatea de adresare este sectorul. Localizarea unui sector se face pe baza următoarelor informații:

— numărul unității (0 sau 1);

— numărul feței (0 sau 1);

— numărul pistei (0-39 la discuri de 360 Ko și 0-79 la discuri de 720 Ko sau 1,44 Mo);

Fig. 4.3. Structura discului MS – DOS.



— număr sector (între 1 și 9 la discurile de 5,25 inch sau între 1 și 18 la discurile de 3,5 inch).

Din punct de vedere structural, discul flexibil gestionat sub MS-DOS conține următoarele zone (fig. 4.3):

Poziție	Lungime	Conținut
3	8 oct.	Identificator sistem de operare
11	2 oct.	Număr octeți/sector
13	1 oct.	Număr sectoare/unitatea de alocare
14	2 oct.	Număr de sectoare pentru bootstrap
16	1 oct.	Număr de copii pentru FAT
17	2 oct.	Număr de intrări în directorul rădăcină
19	2 oct.	Număr sectoare/disc
21	1 oct.	Tip disc
22	2 oct.	Număr de sectoare în FAT
24	2 oct.	Număr de sectoare pe pistă
26	2 oct.	Număr de piste
28	2 oct.	Număr de sectoare rezervate

Fig. 4.4. Informații în sectorul 1

1. Programul încărcător (bootstrap). Acest program este memorat în primul sector (*1*) al pistei 0, față de 0 al oricărui disc. Conținutul său depinde de tipul discului (sistem sau non-sistem). Pentru disc sistem în acest sector se găsește o secvență de program care inițializează procesul de încărcare a sistemului de operare de pe disc. în memoria internă. Pentru discurile non-sistem, în acest sector se găsește o secvență de program care afișează un mesaj ce indică faptul că discul respcetiv nu este disc sistem și în consecință nu poate fi încărcat sistemul de operare. Fișierele sistemului de operare nu ocupă o poziție fixă pe disc, ci o poziție fixă în directorul rdădăcină, lucru care simplifică procesul de încărcare a lor în memoria internă. În cadrul acestui sector sînt și unele informații care descriu tipul discului și versiunea sistemului de operare sub care a fost format (fig. 4.4).

2. Tabela de alocare a spațiului disc (FAT). Tabela de alocare a spațiului disc (FAT — *F*ile *A*location *T*able) este memorată începînd cu sectorul 2, pista 0, față de 0 al oricărui disc. Dimensiunile tabelei depinde de capacitatea discului. Rolul acestei tabele este de a ține evidența spațiului ocupat, liber sau defect, pe disc. Deoarece informațiile conținute de FAT sînt vitale pentru utilizarea sistemului de fișiere, sistemul de operare păstrează două copii identice ale acestei tabele. Alocarea spațiului disc pentru fișiere se face în unități *de alocare*, numite la IBM *cluster*. Numărul de sectoare care formează un cluster depinde de formatul discului. Fiecare intrare în FAT este asociată unui cluster și ocupă 12 biți. Numărătoarea clusterelor începe cu 2, primele două intrări în FAT fiind neutilizate (primul octet din FAT memorează tipul de format al discului respectiv). Pentru fiecare fișier alocat pe disc, în director se memorează numărul primului cluster alocat. Dacă fișierul conține numai o singură unitate de alocare, atunci în intrarea asociată clusterului respectiv se va memora valoarea OFFFH. Dacă fișierul ocupă mai multe clustere, intrarea din FAT corespunzătoare primei unități de alocare conține numărul următorului cluster etc. Se formează astfel un lanț de alocare a cărui ultimă intrare asociată fișierului conține valoarea OFFFH. În cazul discurilor în format 360 Ko, FAT are dimensiunea de 4 sectoare; se folosesc 2 sectoare pe cluster (512 octeți), iar numărul de clustere este de 720. La discurile de 720 octeți, FAT are 10 sectoare (se folosesc 2 sectoare pe cluster, iar numărul de clustere este 1 440).

Fiecare intrare în FAT corespunde unui cluster, memorînd statutul acelei unități de alocare:

— valoarea 0, cînd unitatea de alocare este neutilizată;

— valoarea OFF7H, pentru unitățile marcate ca defecte (la formatare sau cu un program utilizator);

— valoarea cuprinsă între 2 și OFF6H, cînd unitatea de alocare face parte dintr-un lanț corespunzător unui fișier;

— valoarea OFFFH, pentru ultima unitate de alocare dintr-un lanț;

3. Directorul rădăcină (Directory). Directorul rădăcină (*directory*) este utilizat pentru memorarea informațiilor referitoare la fișiere și la alte subdirectoare, dată fiind structura arborescentă pe care o poate avea sistemul de fișiere. Directorul conține cîte o intrare de 32 de octeți pentru fiecare fișier sau subdirector. care conține informații despre acestea (fig. 4.5). În cadrul discului pot exista mai multe tipuri de directoare (rădăcină) ale căror poziție și lungime sînt fixe și subdirectoare care pot avea orice dimensiune, existînd posibilitatea de a fi memorate oriunde pe disc.

Poziție	Lungime	Format	Conținut
0	8	ASCII	nume fișier
8	3	ASCII	extensie
11	1	codificat	atribut
12	10	zero	nefolosit
22	2	număr	ora creare
24	2	număr	data creare
26	2	număr	cluster
28	4	număr	lungime fișier

Fig. 4.5. Conținutul unei linii din director.

Formatul unei intrări este același indiferent de tipul directorului. Cînd se creează un subdirector, se inițializează două intrări ale aceluia:

— una cu numele de fișier „..“ pentru directorul „tată“;

— și unul cu numele „.“ pentru subdirectorul curent.

• *Numele fișierului* apare în primul cîmp al intrării în director. Dacă numele este mai scurt decît opt caractere, se completează cu blancuri la dreapta. Primul octet din nume indică statutul intrării: dacă este zero, intrarea nu a fost utilizată niciodată; dacă octetul are valoarea OE5H, înseamnă că intrarea respectivă a fost utilizată, dar fișierul corespunzător a fost șters. Deoarece în cazul ștergerii unui fișier doar primul octet este codul de valoare OE5H, se poate face ușor recuperarea fișierelor șterse accidental, cu condiția ca spațiul alocat fișierului să nu fi fost între timp reutilizat.

• Cîmpul *extensie* este format din maximum 3 caractere.

• Cîmpul *atribut* ocupă un octet în care sînt codificate caracteristicile fișierului (fig. 4.6):

— invizibil, sistem — fișierul nu este accesibil prin operații DOS obișnuite. De exemplu, numele unui astfel de fișier nu apare afișat ca urmare a execuției unei comenzi DIR; un astfel de fișier nu poate fi copiat prin comanda COPY;

bit								Valoare numerică	Semnificația
7	6	5	4	3	2	1	0		
							1	1	read-only
						1		2	invizibil
					1			4	sistem
				1				8	etichetă volum
			1					16	subdirector
		1						32	arhivă
	1							64	neutilizat
1								128	neutilizat

Fig. 4.6. Atributele unui fișier.

— etichetă volum — semnifică faptul că intrarea respectivă nu se referă la un fișier, ci conține (în primii 11 octeți) eticheta acestui volum. O astfel de intrare poate să apară doar în directorul rădăcină;

— subdirector — intrarea respectivă se referă la un subdirector. Acesta este memorat ca un fișier obișnuit;

— arhivare — acest atribut are semnificație doar pentru discurile Wincester.

* *ora de creare* este un cîmp folosit pentru a memora ora la care a fost creat sau modificat ultima dată fișierul respectiv. Valoarea memorată în acest cîmp se calculează după formula:

ora de creare = ore * 2 048 + minute * 32 + secunde/2;

* *data de creare* memorează data la care a fost creat sau modificat ultima dată fișierul respectiv. Se calculează cu formula:

data = (an — 1984) * 512 + luna * 32 + ziua;

* *număr cluster* conține numărul primei unități de alocare atribuite fișierului respectiv;

* *lungime fișier*, conține numărul de octeți în fișier;

* *fișierele* ocupă spațiul rămas disponibil. Unitățile de alocare se acordă fișierelor treptat, pe măsură ce acestea au nevoie de ele. Algoritmii de alocare sînt complicați, avînd în vedere mai buna recuperare a unor fișiere șterse în mod accidental.

4.2.2. Utilizarea discului Wincester

Din punct de vedere structural, discul Wincester (discul fix) conține aceleași zone ca și discurile flexibile: programul încărcător, tabele de alocare a spațiului disc, directorul rădăcină și spațiul alocat fișierelor. Dată fiind capacitatea lor sporită (în mod tipic 20 Mocteți, 40 Mocteți etc.), discurile fixe pot fi împărțite în una pînă la 4 zone distincte, de dimensiuni diferite, numite partiții. Sistemul de operare poate avea acces la mai multe partiții, recunoscînd două tipuri de partiții:

— *partiție primară*, cea care este necesară pentru a folosi discul fix ca disc sistem (poate avea o dimensiune de pînă la 32 Mocteți);

— *partiție extinsă*, care poate fi de orice dimensiune, și poate fi împărțită în *unități logice*, ce pot avea fiecare pînă la 32 Mocteți.

Ambele tipuri de partiții sînt create cu ajutorul programului FDISK, care permite, în cadrul partițiilor extinse, împărțirea acestora în unități logice.

O partiție poate fi declarată activă (preia controlul) în momentul în care sistemul de operare este lansat de pe discul Wincester. O astfel de lansare se face numai dacă unitatea *A* de disc flexibil nu conține disc. Discurile sînt referite prin litere urmate de două puncte. În mod uzual unitățile de disc flexibile sînt desemnate de sistem prin *A* și *B*, iar discul Wincester prin litera *C*. La lansarea sistemului de operare, fiecărei unități logice din partiția extinsă i se va atribui o literă, permițînd adresarea acesteia la fel ca și în cazul unităților fizice. Pregătirea discului Wincester se face cu ajutorul programului FDISC. Datorită meniurilor este facilitată alegerea operației dorite, care poate fi una din următoarele:

— crearea unei partiții DOS;
— schimbarea partiției active;
— ștergerea unei partiții DOS;
— afișarea informațiilor despre partiții.

Crearea unei partiții. Un disc Wincester este împărțit în cilindri. De exemplu, discul de 10 Mocteți conține 305 cilindri, fiecare cilindru avînd 34 816 octeți. Acest spațiu se poate aloca în proporții dorite partiției primare și partiției extinse. Îm mod opțional, toți cilindri se pot aloca partiției primare, în care se va încărca sistemul de operare. Înainte de folosirea unei partiții astfel create, ea trebuie formatată. Fiecărei partiții create cu comanda FDISC i se va aloca, de către sistemul de operare, o literă de unitate.

Schimbarea partiției active. Această opțiune se folosește cînd se dorește predarea controlului, la pornirea calculatorului unui alt sistem de operare, dintr-o altă partiție de pe discul fix. În acest caz, la lansarea sistemului de operare, acesta va fi încărcat de pe discul indicat ca partiție activă. Partițiile extinse nu pot fi declarate active.

Ștergerea unei partiții. Această operație conduce la pierderea tuturor fișierelor alocate în partiția respectivă. După executarea operației de ștergere a partiției active, este necesară lansarea unui alt sistem de operare de pe o altă unitate.

Afișarea informațiilor despre partiții. Se vor afișa pe ecran, pentru fiecare partiție, următoarele informații:

— numărul partiției (1-4);
— starea partiției (A-active);
— tipul partiției (DOS — pentru partițiile primare, și EXT-DOS, pentru cele extinse);
— numărul primului cilindru al partiției;
— dimensiunea partiției, în cilindri.

După crearea unei partiții DOS primare, pentru a formata partiția și a instala sistemul de operare, se va utiliza comanda SELECT:

SELECT d : [cale]

unde *d* reprezintă litera alocată unității de disc, iar calea reprezintă directorul în care se vor copia comenzile externe (subdirectorul care va fi creat după formatare). La sfîrșit se va apela comanda XCOPY pentru copierea comenzilor externe MS-DOS.

4.3. Sistemul de fișiere la MS-DOS

Așa cum s-a arătat în paragraful precedent, în cadrul sistemului de operare MS-DOS spațiul de memorie al unui disc este împărțit în 4 zone: primele trei zone sînt ale sistemului de operare, iar ultima, care ocupă și cel mai mare spațiu, este zona care conține fișierele. În fiecare unitate de disc fișierele fac parte dintr-o structură arborescentă cu un număr nelimitat de niveluri. Acest lucru se datorește posibilității oferite de sistemul de operare MS-DOS de a grupa fișierele, după dorință, în directoare (fig. 4.7).

În urma formatării unui disc, sistemul creează pe aceasta un singur director, numit *director rădăcină*. Acesta poate conține numele altor fișiere, dar și al altor directoare, care la rîndul lor pot conține fișiere și directoare ș.a.m.d. Spre deosebire de directorul rădăcină, toate celelalte directoare

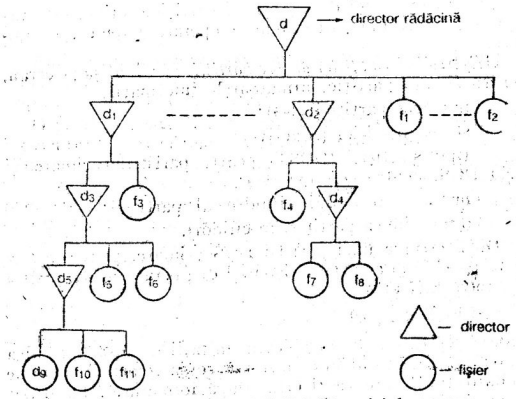


Fig. 4.7. Structura multinivel a sistemului de operare.

se numesc subdirectoare. Directorul rădăcină poate conține 112 intrări (fișiere sau subdirectoare) la discurile flexibile de 360 sau 720 Kocteți sau 512 intrări la discurile fixe. Subdirectoarele sînt în realitate fișiere și ca urmare nu sînt restrînse din punctul de vedere al dimensiunii; ele pot conține oricîte intrări de fișiere și subdirectoare, în funcție de spațiul disponibil existent pe disc.

Sistemul de operare reține informațiile referitoare la fiecare fișier în directoare (caracteristicile fișierului) și în tabela de alocare a spațiului disc FAT (unitățile de alocare afectate fișierului) (fig. 4.8).

Sistemul de operare memorează unitatea de disc implicită cu care se lucrează, precum și *directorul curent*, adică directorul în care sistemul caută fișierele care sînt specificate fără directorul din care fac parte. Schimbarea directorului curent se poate face cu ajutorul comenzii CHDIR. La lansarea sistemului de operare directorul rădăcină se consideră drept director curent.

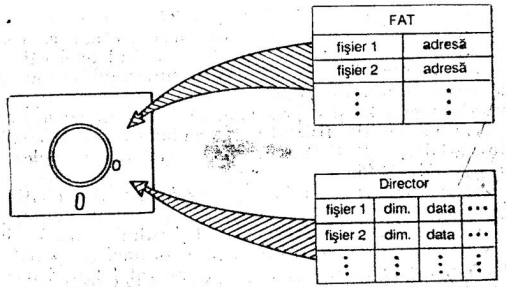


Fig. 4.8. Gestiunea fișierelor în MS–DOS.

4.3.1. Specificarea fișierelor

Pentru a specifica în mod complet un fișier, trebuie indicate unitatea disc pe care se află, directorul în care este creat, precum și numele fișierului. Pentru a indica directorul trebuie folosită o cale, adică o secvență de directoare care să localizeze în mod sigur directorul în care se află fișierul căutat. Specificarea unui fișier în sistemul MS-DOS are următorul format:

[d :] [cale] nume [.extensie]

Specificația cîmpurilor este următoarea:

- *d* — reprezintă unitatea disc, specificată printr-o literă. În mod uzual se folosesc *A* și *B*, pentru unități de disc flexibil, și *C*, *D*, ..., pentru unități sau partiții de disc Winchester;
- *cale* — reprezintă calea către directorul în care se află fișierul specificat. Ea constă dintr-o serie de nume de directoare, separate prin caracterul backslash (\), ce permit localizarea sigură a fișierului în arborele sistemului de fișiere. Există două tipuri de căi: *absolute*, care încep cu/și specifică sistemului MS-DOS să înceapă căutarea din

directorul rădăcină al unității, și *relative*, care încep cu un nume și indică faptul ca, în continuare, căutarea să se facă plecînd din directorul curent. În cadrul unei căi se pot folosi și numele: „." — pentru directorul curent și „..", pentru directorul „tată". Se consideră exemplul prezentat în figura 4.8 și se presupune că directorul curent este d_2. Pentru a specifica fișierul f_6, putem folosi o cale absolută (din rădăcină), /d1/d3/f6, sau o cale relativă, care pleacă din d_3, ../d1/d3/f6.

Pentru a referi fișierul f_8, putem folosi calea relativă d4/f8.

• *Nume* — reprezintă numele fișierului compus din 1—8 caractere, care pot fi litere, cifre și caractere speciale: ,@ , $#, , %,f , (,), —, , , /,\. Se pot folosi și două caractere speciale — *metacaractere* —, care permit specificarea mai multor fișiere într-o singură comandă:

— caracterul „?" într-un nume de fișier sau extensie indică faptul că orice caracter poate ocupa această poziție. De exemplu, TEST .PAS poate specifica TEST1 PAS sau TESTA.PAS etc.;

— caracterul „*" într-un nume de fișier sau extensie de fișier indică faptul că orice caracter poate ocupa acea poziție și tot restul pozițiilor în numele de fișier. De exemplu, TEST * .PAS semnifică specificarea oricărui fișier al cărui nume începe cu șirul TEST (de exemplu, TEST1.PAS sau TEST100.PAS). Dacă se folosește *.*, acest șir specifică faptul că se dorește referirea la toate fișierele de pe disc. Există cîteva *nume rezervate* pe care sistemul de operare le folosește pentru desemnarea unor periferice din configurație. Ca urmare, aceste nume nu pot fi folosite pentru desemnarea unor fișiere:

— *aux*, *com*1 — primul port al adaptorului de comunicații asincron;

— *con* — tastatura consolei (la intrare) sau ecranul consolei (la ieșire);

— *prn*, *lpt*1 — prima imprimantă paralelă;

— *lpt*2, *lpt*3 — alte imprimiante paralele;

— *com*2, *com*3, *com*4 — alte porturi ale adaptorului asincron;

— *null* — perifericul fictiv, utilizat pentru testarea unor programe, simulînd echipamentul de intrare (generează imediat sfîrșit de fișier) sau echipamentul de ieșire (fără să scrie în mod real nici o informație);

• *extensia* — reprezintă numele extensiei fișierului, compusă din 1-3 caractere. Specificarea extensiei este opțională la crearea fișierului, fiind însă obligatorie în cazul referirii unui fișier care a fost creat cu extensie.

4.4. FORMATUL PROGRAMELOR EXECUTABILE

Sistemul de operare MS-DOS admite două formate de memorare a programelor pe suport extern pentru programele executabile: formatul EXE și formatul COM. În cazul formatului EXE fișierul corespunzător conține programul într-un format relocabil și un antet cu informațiile necesare procesului de relocare. În cazul formatului COM, fișierul conține imaginea memorie a programului. Ca urmare, spațiul ocupat pe disc de către programele în format COM este mai redus (cu mai mult de 758 octeți), iar încărcarea în memorie și lansarea lor în execuție se fac mai rapid. Cu toate acestea formatul COM este mai puțin utilizat datorită unor restricții pe care le impune. La lansarea unui program acesta va fi încărcat la o adresă care succede zona de memorie ocupată în programul COMMAND.COM și orice alte programe care au fost încărcate și lăsate rezidente în memorie. În momentul lansării, programului i se alocă întreaga memorie disponibilă. În memorie un program încărcat pentru execuție este precedat de un prefix format din 256 de octeți. Acest prefix are o structură fixă indiferent de formatul programului încărcat. Structura prefixului (PSP) este prezentată în fig. 4.9.

Adresă relativă	Dimensiune	Semnificație
0	2 oct.	cod instrucțiune de terminare
2	2 oct.	Adresa sfîrșitului memoriei exprimată în multiplii de 16 octeți
4	1 oct.	Rezervat
5	5 oct.	Instrucțiune de apel de procedură spre punctul de intrare în DOS corespunzător funcțiilor DOS
10	4 oct.	Adresa de revenire în sistemul de operare la sfîrșitul execuției programului
14	4 oct.	Adresa de tratare CTRL/BREAK
18	4 oct.	Adresa de tratare a erorilor critice DOS
22	2 oct.	Rezervat pentru DOS
44	2 oct.	Adresa de segment pentru un șir de caractere care conține o descriere a unor informații de sistem.
46	46 oct.	Rezervat pentru DOS
92	16 oct.	FCB pentru primul parametru
108	20 oct.	FCB pentru al doilea parametru
128	1 oct.	Numărul de opțiuni în linia de apel a programului
129	127 oct.	Linia de apel a programului din care s-au eliminat numele programului și directivele de redirectivare.

Fig. 4.9. Structura prefixului.

4.5 INTERFAȚA UTILIZATOR

Comenzile MS-DOS permit utilizatorului să comunice cu calculatorul. În general, prin intermediul lor se pot face comparări, copieri, afișări, ștergeri de fișiere sau volume, se pot lansa în execuție programe, se poate afișa conținutul unor directoare, se pot obține informații despre calculator și sistemul de operare. Există două tipuri de

comenzi MS-DOS: *interne* și *externe*. Comenzile interne sînt cele mai folosite; ele fac parte dintr-o componentă a sistemului de operare COMMAND-COM — interpretorul de comenzi. Ca urmare, fiind rezidente în memoria internă, se execută imediat. Comenzile externe sînt rezidente pe disc; ca urmare, pentru a fi executate ele trebuie citite de pe disc și încărcate în memorie. Orice fișier .COM, .EXE sau .BAT este considerat ca fiind o comandă externă. Comenzile externe trebuie specificate cu calea corespunzătoare, în caz contrar ele fiind căutate doar în directorul curent. Cu ajutorul comenzii PATH se pot indica și alte directoare suplimentare în care pot fi căutate comenzile externe. Cînd se tastează o comandă externă nu trebuie specificată extensia.

Trebuie reținute cîteva considerații generale care se aplică la toate comenzile:

1) Comenzile sînt urmate de obicei de una sau mai multe opțiuni (parametri). Unii parametri sînt obligatorii, alții opționali. Dacă cei opționali nu sînt specificați, sistemul preia valoarea lor implicită;

2) Comenzile și parametrii pot fi introduși atît cu litere mari, cît și cu litere mici;

3) Comenzile și parametrii trebuie separate prin spații, virgulă, punct și virgulă. Sistemul de operare tratează identic toate aceste caractere ca delimitatori;

4) În cazul parametrilor nume de fișiere, se acceptă și introducerea căii corespunzătoare înaintea fișierului, pentru definirea fără echivoc a poziției fișierului în arbore;

5) Numele unui fișier poate să nu conțină numele extensiei în cazul în care fișierele sînt create sau redenumite. Dacă însă se referă la nu fișier existent care are extensie, acesta trebuie inclus în specificatorul de fișier;

6) Se poate opri execuția unei comenzi utilizînd CTRL/C (CTRL/BREAK);

7) Comenzile devin efective abia după ce s-a apăsat tasta RETURN (ENTER), la sfîrșitul introducerii lor;

8) Atunci cînd o comandă produce o mare cantitate de date la ieșire, pe ecran se poate folosi CTRL/S pentru a suspenda temporar fișierul. Apăsînd apoi orice caracter, afișarea continuă.

4.5.1. Redirectarea fişierelor standard

În mod implicit, MS-DOS presupune că echipamentul standard de intrare este tastatura, iar echipamentul standard de ieşire — ecranul. Ca urmare, comenzile MS-DOS vor fi introduse de la tastatură şi vor produce mesaje care sînt afişate pe ecran. Sistemul permite însă redirectarea acestui flux standard. O comandă poate primi date dintr-un fişier în loc de la tastatură sau poate produce mesaje într-un fişier disc, în loc să le afişeze pe ecran. De asemenea, facilitatea de „pipe" poate face conectarea între două programe, în sensul că datele de ieşire ale unuia să devină date de intrare pentru cel de-al doilea. Acest lucru înseamnă că programe care fac intrări/ieşiri pe echipamentele standard pot fi folosite fără nici o modificare în cazul în care se utilizează drept intrări şi/sau ieşiri fişiere disc.

• *Redirectarea ieşirii* face ca informaţiile pe care programul le afişează în mod uzual pe ecran să fie scrise într-un fişier. Pentru a indica redirectarea ieşirii se foloseşte caracterul „ **>**":

>fişier

De exemplu, comanda **dir** afişează pe ecran conţinutul directorului curent. Dacă vrem să memorăm acest conţinut în fişierul **fis**, vom putea folosi redirectarea la această comandă:

dir > fis

În acest caz, MS-DOS creează fişierul **fis** (dacă nu există), scriind pe el datele de ieşire ale programului **dir.**

Dacă vrem ca informaţia scrisă în urma redirectării fişierului de ieşire să fie adăugată la sfîrşitul unui fişier existent (în loc să scrie de la început) se folosesc două semne„ **> >**". De exemplu **dir > > fis** semnifică faptul că în fişierul **fis** se adaugă conţinutul directorului curent.

• *Redirectarea intrării* face ca toate operaţiile de intrare ale programului să se efectueze dintr-un fişier disc în locul tastării. Pentru a indica acest tip de redirectare se foloseşte semnul „**<**":

<fişier

În cadrul unei comenzi se poate utiliza simultan redirectarea atît a fişierului standard de intrare, cît şi a fişierelor de ieşire. De exemplu, comanda:

Sort < fis1 > fis2

va prelua date din fişierul fis1 pe care le va sorta alfabetic, linie cu linie şi le va scrie în fişierul fis2.

• *Conectarea programelor prin „pipe"* permite ca ieşirea standard a unui program să poată fi utilizată ca intrare standard pentru alt program. Acest lucru înseamnă de fapt o înlănţuire de programe, cu redirectarea automată a intrării/ieşirii. Numele programelor care se înlănţuie prin „pipe" se separă prin caracterul (:). De exemplu:

dir : sort

va avea ca efect afişarea conţinutului directorului curent, sortat în ordine alfabetică. Sistemul de operare MS-DOS utilizează fişiere temporare care se creează în directorul rădăcină al discului implicit cu numele %PIPEn.**$$$**. La sfîrşitul execuţiei lanţului de programe, aceste fişiere sînt şterse automat.

4.5.2. Comenzi MS-DOS

Vom prezenta principalele comenzi recunoscute de MS-DOS, nu în ordine alfabetică, ci în funcţie de rolul pe care-l joacă în gestiunea resurselor sistemului de operare. Vom folosi cîteva metasimboluri cunoscute, cum ar fi (...), pentru a indica repetarea unor parametrii, ([ ]) pentru a indica parametrii opţionali sau (< >)pentru a defini generic cîmpuri care vor fi înlocuite de şiruri de caractere după dorinţa utilizatorilor. La fiecare comandă vom nota în paranteză dacă este internă sau externă.

4.5.2.1. *Comenzi cu privire la directoare*

1. CD (internă)

Funcţia — permite schimbarea directorului curent sau afişarea acestuia.

Sintaxa — *cd*[<d: >] [<cale >] sau *chdir* [<d: >] < [<cale >]

Dacă nu se specifică discul (d) se consideră unitatea implicită. De exemplu, comanda cd ... va schimba directorul curent cu directorul „tată" al vechiului director curent.

Dacă se folosește comanda cd fără parametru, se afișează calea pînă la directorul curent.

Comanda cd a: va afișa directorul curent al discului A.

2. DIR (internă)

Funcția — listează fișierele dintr-un director

Sintaxa — *dir* [⟨d: ⟩] [⟨cale ⟩] [/p] [/w]

Dacă se tastează doar dir, se listează conținutul directorului curent de pe perifericul implicit. În informația afișată se includ numele volumului disc, mărimea spațiului liber rămas și cîte o linie corespunzătoare fiecărui fișier, incluzînd dimensiunea fișierului, data și ora la care s-a făcut ultimul acces la fișier. Intrările corespunzătaore altor subdirectoare sînt marcate prin șirul ⟨dir ⟩. Parametrul /p are ca efect execuția unei pauze ori de cîte ori ecranul este plin cu informația afișată. După apăsarea oricărei taste, afișarea continuă. Parametrul /w produce o afișare condensată a informațiilor din director (doar numele fișierelor și al subdirectoarelor). În precizarea căii se pot omite numele fișierului și extensia. În acest caz sistemul presupune existența caracterului * în locul lor. De exemplu comanda:

dir c:/biblio — va lista tot conținutul directorului biblio de pe discul c;

dir/biblio/prog — va lista informații despre toate fișierele prog din directorul biblio, indiferent de extensia lor (se consideră implicit prog.*)

3. MD (internă)

Funcția — permite crearea unui nou subdirector

Sintaxa — *md* [⟨d: ⟩] ⟨cale ⟩ sau *mkdir*[⟨d : ⟩] ⟨ ⟨cale ⟩

Această comandă permite crearea unei structuri arborescente în sistemul de fișiere. În fiecare director se pot crea subdirectoare, care la rîndul lor pot conține alte subdirectoare etc. Singura limită în crearea unor noi subdirectoare

este spațiul disc disponibil. De exemplu, comanda md/clasa|dan va crea un nou subdirector, *dan*, în directorul *clasa* situat în primul nivel sub rădăcină.

4. RD (internă)

Funcția — Șterge director

Sintaxa — *rd* [⟨ d: ⟩] ⟨cale ⟩ sau[⟨d: ⟩] ⟨cale ⟩

Cu ajutorul acestei comenzi se șterge un subdirector indicat prin cale. Directorul trebuie să fie vid, adică să conțină doar intrările și ..; de menționat că directorul rădăcină și directorul curent nu pot fi șterse.

5. JOIN (externă)

Funcția — Conectează o unitate disc la un director de pe o altă unitate disc pentru a produce o singură structură de director din cele două discuri.

Sintaxa — *join* ⟨d1: ⟩ ⟨d2: ⟩ ⟨cale ⟩[/d]

Prima unitate d1 indică discul care va fi conectat în directorul indicat de cale de pe discul d2. Directorul indicat de cale trebuie să fie subdirector al rădăcinii (primul nivel de adîncime). Dacă directorul specificat nu există, el este creat; dacă există trebuie să fie gol. Parametrul /d face deconectarea unui disc de la un director (operațiune complementară comenzii join). Să considerăm spre exemplu că pe discurile A: și B: există structuri de fișiere din figura 4.10. În urma execuției comenzii join B: A:\d2, din cele două sisteme se va crea un singur arbore de fișiere, organizat pe cele două unități, ca în figura 4.11.

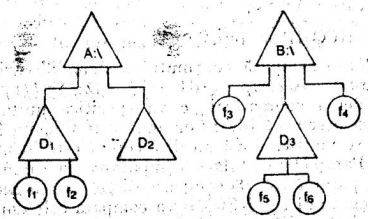


Fig. 4.10. Structuri de fișiere înainte de conectare.

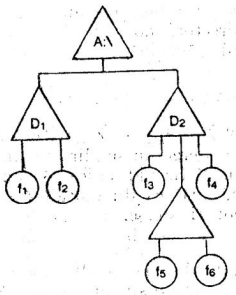


Fig. 4.11. Structura de fișier după execuția comenzii JOIN.

4.5.2.2. *Comenzi cu privire la volume*

1. ASSIGN (externă)

Funcția — Asignează o literă unui periferic
Sintaxa — *assign* [[x] = [Y]]

Această comandă se utilizează cînd se dorește dirijarea tuturor operațiilor cu unitatea de disc spre o altă unitate de disc. Numele primei unități, desemnat de litera *x*, va fi înlocuit intern de sistemul de operare cu numele celei de-a doua unități, desemnată de litera *y*. De acum înainte toate cererile de intrare/ieșire pentru unitatea *y* se vor reorienta către unitatea *x*. De exemplu, comanda ASSIGN A = B va face ca intrările/ieșirile adresate lui B: să fie dirijate către A:.

2. DISKCOMP (externă)

Funcția — compară conținutul a două discuri flexibile
Sintaxa — *diskcomp* [⟨ d1 ⟩:][⟨d2: ⟩] [/1] [/8]

În vederea comparării se va specifica unitatea sursă (d1) și cea destinată (d2). Comanda este folosită în special pentru a se verifica dacă un disc copiat este identic cu cel asignat. Parametrul /1 face compararea la discuri simplă față, iar parametrul /8 compară discuri cu 8 sectoare pe pistă. Programul DISKCOMP va compara cele două discuri pistă cu pistă, afișînd mesaj de eroare în cazul în care conținutul este diferit; în acest caz ne indică numărul pistei și fața unde s-a găsit neconcordanța. Dacă se omit ambii parametri care indică unitățile de disc, se va executa o comparare cu o singură unitate implicită. Se va monta mai întîi primul disc care va fi citit și apoi se va schimba discul, efectuîndu-se compararea propriu-zisă.

3. DISKCOPY (externă)

Funcția — copiază conținutul unui disc flexibil sursă pe un disc flexibil destinație.
Sintaxa — *diskcopy* [⟨d1: ⟩] [⟨d2: ⟩]

Primul parametru (d1) indică discul sursă, iar cel de-al doilea (d2) discul destinație. Se poate specifica aceeași unitate de disc, caz în care se caută prin mesaj momentul cînd trebuie schimbat discul. În cazul în care discul destinație nu este formatat, programul DISKCOPY va formata acest disc inainte de copiere. Copierea se face sector cu sector, începînd cu primul, astfel încît eventualele spații libere dintre fișiere se vor păstra în discul destinație. Pentru aliminarea acestei fragmentări a spațiului alocat se indică faptul ca în continuare copierea să se facă cu utilizarea comenzii COPY, fișier cu fișier.

4. VERIFY (internă)

Funcția — Activează sau dezactivează funcția de verificare a scrierii pe disc.
Sintaxa — *verify* [on/off]

Atunci cînd funcția de verificare este selectată (*on*) sistemul de operare execută verificare după fiecare operație de scriere pe disc, analizînd dacă datele scrise pot fi citite fără eroare. Este evident că selecția acestei funcții face ca sistemul să lucreze mai lent, existînd cîte două accese la fiecare scriere pe disc.

5. FORMAT (externă)

Funcția — formatează (inițializează) discul pentru a accepta fișiere MS-DOS.
Sintaxa — *format* ⟨d: ⟩[/1][/8][/s][/v][/b]

Comanda are rolul de a inițializa un disc *distrugînd toate informațiile* care eventual existau pe el. Se creează

tabele de alocare a spațiului (FAT); directorul rădăcină și eventual se adaugă încărcătorul sistemului. Parametrul /1 cere formatarea discului în simplă față (implicit formatarea se face în dublă față). Parametrul /8 cere ca formatarea să se facă cu 8 sectoare /pistă (implicit, 9 sectoare/pistă). Perimetrul /b cere ca formatarea să se facă cu 8 sectoare /pistă și să se rezerve spațiu pentru fișierul sistemului de operare; IBMBIO.COM, IBMDOS.COM, fără ca cle să fie plasate pe disc. Acest lucru se va face ulterior cu ajutorul unei comenzi SYS. Parametrul /v cere introducerea în directorul rădăcină a unei etichete de volum (maxim 11 caractere) Parametrul /s specifică faptul că după procesul de formatare să fie copiate și fișierele sistemului de operare IBMBIO.COM, IBMDOS.COM, și COMMAND-COM. În timpul procesului de formatare toate pistele defecte sînt marcate în FAT ca rezervate pentru a se împiedica alocarea lor la fișiere. Orice disc nou, înainte de utilizare, trebuie formatat pentru a i se crea structura de organizare a fișierelor recunoscute de MS-DOS.

6. SYS (externă)

Funcția — transferă fișierele sistem MS-DOS de pe discul implicit pe discul specificat.

Sintaxa — sys ⟨d: ⟩

Comanda este necesară pentru transferarea fișierului IBMBIO.COM și IBMDOS.COM pe un disc sistem. Fișierul COMMAND.COM va trebui transferat cu comanda *copy*. Directorul discului la care se aplică comanda *sys* trebuie să fie vid sau discul să fi fost formatat cu opțiunea /b pentru a se rezerva spațiu pentru aceste fișiere. Acest lucru este necesar deoarece la lansarea sistemului de operare el trebuie să găsească cele două fișiere în primele două intrări ale directorului rădăcină.

7. SELECT (internă)

Funcția — Permite instalarea sistemului de operare pe un disc nou.

Sintaxa — select [a:][b:][⟨d: ⟩] [⟨cale ⟩]

Comanda *select* se folosește de către comand *format* pentru a formata discul destinație (⟨d ⟩). Apoi, cu comanda *xcopy* se copiază toate comenzile externe ale sistemului de

operare. În sfîrșit, se creează fișierul CONFIG.SYS (pentru configurarea sistemului) și AUTOEXEC.BAT (fișiere de comenzi indirecte executate la inițializarea sistemului de operare). Ca discuri sursă, de unde se face copierea, sînt permise doar a: sau b:.

8. VOL (internă)

Funcția — Afișează, dacă există, eticheta volumului (discului) indicat.

Sintaxa — vol [⟨d: ⟩]

9. LABEL (externă)

Funcția — creează, șterge sau schimbă eticheta de identificare a unui volum disc.

Sintaxa — label [⟨d: ⟩] [⟨eticheta ⟩]

Eticheta poate avea cel mult 11 caractere. Dacă eticheta nu este precizată în cadrul comenzii, sistemul cere această etichetă. Dacă se tastează ENTER, sistemul întreabă dacă se dorește ștergerea vechii etichete.

4.5.2.3. *Comenzi cu privire la fișiere*

1. *COPY* (internă)

Funcția — copiază unul sau mai multe fișiere pe un anumit disc.

Sintaxa — *copy* [⟨d1: ⟩] ⟨cale ⟩[⟨d2: ⟩][⟨cale ⟩] [/v] [/a] [/b] (pentru copiere)

copy ⟨cale ⟩ + ⟨cale ⟩ + ... + + ⟨cale ⟩ (pentru concatenare)

Primul fișier specificat este fișierul sursă (cel care va fi copiat). Cel de-al doilea este fișierul destinație. Dacă al doilea parametru este un director, fișierele sînt copiate fără schimbarea numelui. Parametrul /v cere sistemului de operare să verifice dacă sectoarele scrise pe discul destinație sînt înregistrate corect. Parametrii /a și /b pot fi folosiți atît de fișierul sursă, cît și de cel destinație. Fiecare se aplică fișierului care-l precede și la toate fișierele din comandă, pînă la întîlnirea altui parametru /a sau /b.

Cînd sînt folosiți cu fișierul sursă:

/a — determină tratarea fișierului ca un fișier ASCII (text). Datele sînt copiate doar pînă la întîlnirea lui EDF (CTRL/Z), exclusiv acest caracter;

/b — determină ca întregul fișier să fie copiat, inclusiv EOF.

Cînd sînt folosiți cu fișierul destinație:

/a — determină adăugarea lui EOF la sfîrșitul fișierului.

/b — nu se adaugă EOF la sfîrșitul fișierului.

De exemplu, comanda copy f1.txt /a f2. txt va determina copierea fișierului f1, pînă la primul EOF, exclusiv acest caracter.

Comanda copy f1.* b: va determina copierea tuturor fișierelor cu numele f1, indiferent de extensie, pe discul b:; după copierea fișierelor destinație, vor purta același nume cu fișierul sursă. În cazul operațiilor de copiere pot fi folosite și nume rezervate, ca de exemplu:

```
copy con f1.txt
copy con aux
copy aux f2.txt
copy f3.txt lpti
```

Prima din aceste comenzi permite utilizarea lui *copy* pentru introducerea rapidă a unor date într-un fișier text. Sfîrșitul introducerii se marchează cu CTRL/Z. În a doua variantă sintactică comanda *copy* permite conectarea unor fișiere în timpul copierii. În acest fel pot fi combinate două sau mai multe fișiere într-unul singur prin adăugarea fișierelor suplimentare la sfîrșitul primului. De exemplu, comanda *copy* p1.pas+p2.pas+p3.pas p.pas va determina combinarea celor trei fișiere sursă într-unul singur, care va purta numele *p.pas*.

2. *XCOPY* (externă)

Funcția — copiază grupuri de fișiere și directoare, inclusiv subdirectoare ale acestora.

Sintaxa-*xcopy* [⟨d1: ⟩] ⟨cale ⟩ [⟨d2: ⟩] [⟨cale ⟩] [/a] [/d:mm-dd-yy] [/e] [/m] [/p] [/s] [/v]

Primul disc și prima cale specifică fișierele sau directoarele care vor fi copiate. Ce-a de-a doua cale indică unde se va face copierea; dacă lipsește, se consideră directorul

implicit, fișierele păstrîndu-și numele după copiere. Parametrii au următoarea semnificație:

/a — determină copierea doar a acelor fișiere care au bitul de arhivare poziționat;

/d — determină copierea doar a fișierului modificat după data indicată;

/e — permite copierea și a subdirectoarelor goale; astfel acestea nu mai sînt copiate;

/m — determină copierea doar a fișierelor care au bitul de arhivare poziționat, anulînd după copiere acest bit;

/p — întreabă utilizatorul înaintea copierii fiecărui fișier;

/v — determină verificarea scrierii fiecărui fișier pe discul destinație;

/s — permite copierea fișierelor din directorul sursă și din toate subdirectoarele de pe nivelurile inferioare ale acestora. De exemplu, comanda *xcopy* a: b: /s /e va determina copierea tuturor fișierelor și a subdirectoarelor, inclusiv a celor goale, de pe discul a: pe discul b:

3. *BACKUP* (externă)

Funcția — Permite salvarea unuia sau mai multor fișiere de pe un disc pe altul.

Sintaxa — *backup* ⟨d1: ⟩ [⟨cale ⟩] ⟨d2: ⟩ [/s] [/m] [/a] [/d:mm-dd-yy] [/t:hh:mm:ss] [/f]

Se pot face salvări între toate tipurile de discuri (flexibile și Winchester), ținînd cont de capacitatea lor diferită. Se face salvarea grupului de fișiere sursă (pe d1) pe unitatea destinație (d2). Fișierele salvate cu *backup* vor putea fi utilizate doar după ce vor fi restaurate cu comanda *restore*.

Semnificația parametrilor utilizați este următoarea:

/s — determină salvarea tuturor fișierelor din toate subdirectoarele din nivelurile inferioare directorului curent;

/m — determină salvarea doar a fișierelor modificate de la ultima salvare;

/a — adăugare de fișiere la cele existente;

/d — determină salvarea fișierelor modificate după data indicată;
/t — determină salvarea fișierelor modificate după ora specificată;
/f — determină formatarea prealabilă a discului destinație (d2).

De exemplu, comanda backup /noi/*.* a: /a/m va salva toate fișierele din directorul nou modificate după ultima salvare, de pe discul implicit (fix) pe discul flexibil a:, adăugîndu-le la fișierele existente. Vom observa însă că pentru salvarea integrală a unui disc Winchester de 10 Mocteți sînt necesare 25 discuri flexibile.

4. *RESTORE* (externă)

Funcția — restaurează unul sau mai multe fișiere de pe un disc pe altul.

Sintaxa — *restore* ⟨d1: ⟩ [⟨d2: ⟩] [⟨cale ⟩] [/s] [/p] [/b: mm-dd-yy] /a: [mm-dd-yy] [/m] [/n]

Cu ajutorul acestei comenzi se restaurează fișierele salvate în prealabil cu *backup*. Discul d1: reprezintă sursa, iar discul d2: reprezintă discul destinație pe care se face restaurarea. Semnificația parametrilor este următoarea:

/s — determină restaurarea fișierelor din toate subdirectoarele directorului curent;
/p — determină un dialog înaintea restaurării fiecărui fișier care a fost modificat de la ultima salvare. În acest fel se permite o restaurare selectivă a fișierelor;
/a — permite restaurarea fișierelor modificate după data specificată;
/b — permite restaurarea fișierelor modificate înainte de data specificată;
/m — permite restaurarea fișierelor modificate de la ultima salvare;
/n — permite restaurarea fișierelor care nu mai există pe discul destinație.

De exemplu, comanda restore a: c:*.*/s va realiza restaurarea de pe discul flexibil a: a tuturor fișierelor pe discul c:, în rădăcină și în subdirectoarele sale.

5. *COMP* (externă)

Funcția — permite compararea unor fișiere.

Sintaxa — *comp* [⟨d1: ⟩] ⟨cale ⟩ [⟨d2: ⟩] ⟨cale ⟩

În mod uzual această comandă se folosește pentru compararea unor fișiere în urma copierii pentru a se verifica corectitudinea acestei operații. De exemplu, comanda: comp a: f1.* b: va compara toate fișierele cu numele f1 de pe discul a: cu cele cu același nume de pe discul b:.

6. *RENAME* (internă)

Funcția — permite schimbarea numelui unui fișier specificat de primul parametru cu numele specificat de al doilea parametru.

Sintaxa — *rename* [⟨d: ⟩] ⟨cale ⟩ ⟨cale ⟩

7. *DELETE* (internă)

Funcția — permite ștergerea unui fișier.

Sintaxa — *del* [⟨d: ⟩] ⟨cale ⟩ sau *erase* [⟨d: ⟩] ⟨cale⟩

8. *TYPE* (internă)

Funcția — permite afișarea pe display a conținutului unor fișiere.

Sintaxa — *type* [⟨d: ⟩] ⟨cale ⟩

Această comandă este folosită pentru vizualizarea conținutului unor fișiere text.

9. *PRINT* (externă)

Funcția — permite tipărirea unor fișiere text la imprimantă în paralel cu execuția altor programe pe calculator.

Sintaxa — *print* [⟨d: ⟩] [⟨cale ⟩] [/d: ⟨disp ⟩] [/q: ⟨valoare ⟩] [/t] [/c] [/p]

Sistemul de operare organizează un fir de așteptare cu toate fișierele care urmează să fie listate la imprimantă. Semnificația parametrilor utilizați este următoarea:

/d — specifică dispozitivul de listare; se va considera implicit PRN;
/q — specifică numărul de fișiere pe care le poate avea firul de așteptare (maxim 32); implicit se consideră valoarea 10;

/t — specifică ştergerea tuturor fişierelor care sînt în firul de aşteptare. Dacă un fişier este în curs de listare, listarea se opreşte;

/c — permite selectarea unor fişiere din firul de aşteptare a căror listare se abandonează;

/p — permite adăugarea unor fişiere în firul de aşteptare la listare.

De exemplu, comanda print/t va goli firul de aşteptare, anulînd toate fişierele, iar comanda print a:f1.txt/c f2.txt/p f3.txt va determina ştergerea din fir a fişierului f1 şi adăugarea în coadă a fişierelor f2 şi f3.

10. *RECOVER* (externă)

Funcţia — Determină recuperarea unor fişiere de pe discul care are sectoare defecte.

Sintaxa — *recover* [⟨d:.] sau *recover* ⟨d: ⟩[⟨cale ⟩]

Comanda este utilă pentru recuperarea unor fişiere care au sectoare defecte, mai puţin datele din aceste sectoare. Se poate efectua şi recuperarea tuturor fişierelor de pe disc, în cazul în care directorul rădăcină nu mai poate fi util (prin varianta de sintaxă).

11. *ATTRIB* (externă)

Funcţia — Permite afişarea sau modificarea atributelor unor fişiere.

Sintaxa — *attrib* [+r;—r[[+a;—a] [⟨d:⟩] ⟨cale⟩

Parametrul +r permite protecţia unui fişier la scrierer (*read-only*). Cu —r se conferă dreptul la scriere. Parametrul +a determină poziţionarea pe 1 a bitului de arhivare. În cazul comenzilor BACKUP, RESTORE, XCOPY cu parametrul /m se vor copia doar fişierele avînd acest bit de arhivare pe 1.

12. *EX2BIN* (externă)

Funcţia — converteşte fişierele. EXE care nu au segmente fixate la o formă compatibilă cu programele. COM imagine de memorie. Se realizează astfel o economie de spaţiu disc şi o încărcare mai rapidă a programului.

Sintaxa — *ex2bin* [⟨d1:⟩] ⟨cale⟩ [⟨d2:⟩] [⟨cale⟩]

Prima cale indică fişierul (fişierele) în format. EXE, iar a doua numele fişierelor. COM obţinute.

4.5.2.4. *Comenzi filtru*

1. *MORE* (externă)

Funcţia — este un „filtru" care citeşte date de la echipamentul standard de intrare şi le trimite pe echipamentul standard de ieşire, cîte un ecran la un moment dat.

Sintaxa — *more*

După umplerea unui ecran, se opreşte afişarea, aşteptîndu-se tastarea oricărei taste. În acest fel se face listarea ecran cu ecran a unui fişier. Această comandă se utilizează în „pipe" cu comanda *type*, ca în exemplul:

type f. txt: more

Efectul este acela al afişării fişierului f.txt ecran după ecran, permiţîndu-se astfel citirea informaţiilor aflate pe display.

2. *SORT* (externă)

Funcţia — citeşte date de pe echipamentul standard de intrare, le sortează şi le afişează pe echipamentul standard de ieşire.

Sintaxa — *sort* [/r] [/+n]

Sortarea se face alfabetic crescător, în ordinea codurilor ASCII, la nivel de linie text. Dacă se indică parametrul /r, sortarea se va efectua în ordine descrescătoare. Folosirea parametrului /+n permite ca sortarea să înceapă conform caracterelor din poziţia *n* (implicit, cheia de sortare se consideră că începe în coloana 1). De exemplu, comanda sort/ /r ⟨f1.txt⟩ f2.txt va avea ca efect sortarea fişierului f1 în ordine descrescătoare, rezultatul fiind scris în fişierul f2.

3. *FIND* (externă)

Funcţia — selectează dintre liniile fişierului specificat în linia de comandă pe acelea care conţin un şir specificat, afişîndu-le pe echipamentul standard de ieşire.

Sintaxa — *find* [/v] [/c] [/n] ⟨„şir"⟩ [⟨d:⟩] [⟨cale⟩]

Prin ⟨cale ⟩ se specifică fişierul ale cărui linii vor fi „filtrate"; în cazul cînd nu se specifică nici un fişier, se consideră echipamentul standard de intrare. Prin „şir" se

indică şirul de caractere pe care trebuie să le conţină liniile ce vor fi afişate. Parametrii au următoarea semnificaţie:

/v — determină afişarea liniilor care nu conţin şirul indicat;

/c — determină afişarea doar a unui contor cu numărul de linii ce conţine şirul indicat;

/n — determină numerotarea relativă a liniilor ce vor fi afişate;

De exemplu, comanda dir :find/v „pas" va determina afişarea din directorul rădăcină a discului a: doar a liniilor cu fişiere care conţin şirul PAS printre informaţiile afişate.

4.5.2.5. *Comenzi pentru definirea unor opţiuni*

1. *BREAK* (internă)

Funcţia — setează verificarea caracterului CTRL/C.

Sintaxa — *break* [on : off]

Acest caracter, CTRL/C (CTRL/BREAK), permite întreruperea unui program în curs de execuţie. Prin comanda *break* se poate seta ca verificarea tastării acestui caracter să se facă la fiecare apel de funcţie sistem.

2. *CTTY* (internă)

Funcţia — permite schimbarea perifericului standard de ieşire/intrare.

Sintaxa — *ctty* ⟨periferic ⟩

Implicit, se declară echipament standard pentru introducerea comenzilor consolă (con). Dar o comandă, ca de exemplu ctty aux determină sistemul de operare să utilizeze echipamentul conectat la *aux* (un alt display) drept consolă standard.

3. *PROMPT* (internă)

Funcţia — Defineşte un nou prompter sistem.

Sintaxa — *prompt* [⟨text ⟩]

Această comandă permite schimbarea prompter-ului sistem (de exemplu, c ⟩). Şirul ⟨text ⟩ din linia de comandă este interpretat de sistem ca fiind noul prompter.

Dacă nu se referă nici un text, se va face revenirea la prompterul implicit. Se pot folosi în ⟨text ⟩ şi metaşiruri, sub forma: $c, unde *c* este un caracter dintre următoarele:

$ — caracterul dolar;
t — ora;
d — data;
p — calea pînă la directorul curent;
n — litera discului implicit;
g — caracterul ⟩
l — caracterul ⟨
q — caracterul =
b — caracterul :

De exemplu, comanda prompt pg va genera un prompter cu calea şi caracterul ⟩, cum ar fi B: \noi ⟩.

Comanda prompt ora=$t data=$d va genera un prompter în care se afişează data şi ora, ca de exemplu:

ora = 17 : 04 data = 25 — 06 — 90

4. *SET* (internă)

Funcţia — determină echivalarea unui şir de caractere cu un alt şir pentru utilizare în celelalte programe.

Sintaxa — *set* [⟨sir1 = sir2 ⟩]

Asemenea echivalări de şiruri sînt inserate într-o zonă a interpretorului de comenzi, fiind astfel disponibile tuturor comenzilor şi aplicaţiilor. În acest mod este posibilă introducerea de cuvinte cheie şi parametrii care nu au nici un sens pentru MS-DOS, dar care pot fi găsiţi şi interpretaţi de programele ce examinează zona interpretorului de comenzi. Dacă într-o comandă set nu se specifică sir2, se şterge vechea asignare a lui sir 1. În fine, dacă se tastează doar set, se afişează toate asignările care sînt în curs.

5. *PATH* (internă)

Funcţia — setează căile pentru căutarea comenzilor (fişiere cu extensiile .EXE, .COM, .BAT).

Sintaxa — *path* [⟨d1:⟩] [⟨cale⟩]; [⟨d:⟩] [⟨cale⟩]

În mod normal, sistemul de operare caută comenzile numai în directorul curent al discului implicit. Prin comanda *path* se poate specifica o listă de directoare suplimentară, în care să se procedeze la căutarea comenzilor. Elementele

listei sînt directoare separate prin punct și virgulă. De exemplu, comanda path\; /sys determină căutarea comenzilor în directorul discului implicit și în subdirectorul /sys.

6. *APPEND* (externă)

Funcția — setează căile pentru căutarea fișierelor de date cu orice extensie.

Sintaxa — *append* [⟨d: ⟩] [⟨cale⟩]; [⟨d: ⟩] [⟨cale⟩]

Comanda are o funcție similară cu *path*, cu deosebirea că aici se indică sistemului de operare să caute fișierele cu orice extensie (nu numai .EXE,.COM,.BAT) în mai multe directoare indicate în listă. De exemplu, secvența de comenzi

```
path      a:/aplicație
append    a:/aplicație
```

dă posibilitatea ca programele și fișierele din directorul aplicație să fie găsite fără specificarea căii, indiferent de directorul în care se lucrează.

7. *MODE* (externă)

Funcția — stabilește modul de operare pentru anumite echipamente: imprimanta, adaptorul de comunicații asincron, adaptorul monitor color/grafic.

Sintaxa

— pentru *imprimanta*: *mode* LPTi [:] [n] [m]
unde *i* — reprezintă numărul imprimantei (1, 2, 3);
n — numărul de caractere pe linie (80 sau 132);
m — numărul de linii pe inch (6 sau 8);
— pentru *adaptorul de comunicații asincron*:
mode COMj [:] ⟨viteza⟩ [,⟨paritatea⟩[,[⟨biti-date ⟩], [⟨biti stop ⟩]
unde j — reprezintă numărul adaptorului asincron (1, 2, 3, 4);
⟨viteza ⟩ — 110-19 200 biți pe secundă;
⟨paritatea ⟩ — N (fără paritate), 0 (paritate impară), E — (paritate pară);

⟨biți date ⟩ — 7 sau 8;
⟨biți stop ⟩ — 1 sau 2;
— pentru *redirectarea imprimantei paralele la adaptorul asincron*: *mode* LPTi [:] = COMj [:]
— pentru *setarea modului de lucru al displayului* *mode* n
undë n poate fi: 40-40 caractere/linie;
80-80 caractere/linie;
BW40-alb/negru, 40 caractere/linie;
BW80-alb/negru, 80 caractere/linie;
CO40-color, 40 caractere/linie;
CO80-color, 80 caractere/linie.

4.5.2.6. *Comenzi informaționale*

1. *TIME* (internă)

Funcția — permite introducerea sau afișarea orei curente.

Sintaxa — *time* [hh:mm [:ss.xx]]
unde hh — reprezintă ora;
mm — minutul;
ss — secundele;
xx — sutimile de secundă.

2. *DATE* (internă)

Funcția — permite introducerea sau afișarea datei.

Sintaxa — *date* [mm-dd-yy)]
unde mm — reprezintă luna (număr);
dd — ziua;
yy — anul.

3. *VER* (internă)

Funcția — afișează versiunea sistemului de operare folosit.

Sintaxa — *ver*

4. *CLS* (internă)

Funcția — șterge ecranul echipamentului de ieșire.

Sintaxa — *cls*

5. *CHKDSK* (externă)

Funcţia — actualizează directorul şi tabela de alocare (FAT) a discului specificat şi afişează un raport cu starea discului.

Sintaxa — *chkdsk* [⟨d: ⟩] [⟨cale ⟩] [/f] [/v]

Parametrul /f specifică faptul că se doreşte să se efectueze corecţia erorilor găsite în director sau în tabela de alocare.

Parametrul /v va afişa informaţii detaliate pe parcursul execuţiei comenzii. În raportul afişat se indică spaţiul total al discului, numărul de fişiere, numărul de directoare, spaţiul liber disponibil etc.

4.6. FIŞIERE DE COMENZI

Fişierele de comenzi („batch-files") reprezintă fişiere care conţin comenzi MS-DOS. Adesea, în loc să se tasteze în mod repetat aceeaşi secvenţă de comenzi, este preferabil ca aceasta să fie memorată într-un fişier şi executată simplu, prin tastarea numelui acestui fişier. Un astfel de fişier, numit *de comenzi*, se poate crea fie cu editorul de text EDLIN fie cu comanda *copy*.

Apelul unui fişier de comenzi se face prin:

[⟨d: ⟩] [cale] ⟨nume-fişier-comenzi ⟩ [.BAT] [⟨parametri ⟩]

Toate fişierele de comenzi trebuie să poarte extensia .BAT (de la „batch"). La lansarea în execuţie a unui fişier de comenzi nu este necesar să se introducă şi extensia .BAT, ci doar numele fişierului. Ultima comandă a unui fişier de comenzi poate fi numele altui fişier de comenzi, ceea ce implică saltul în vederea execuţiei în noul fişier de comenzi. Sistemul de operare afişează comenzile din fişier pe parcursul execuţiei lor. Pentru a inhiba însă afişarea unei linii din fişier, ea trebuie precedată de caracterul C. Se poate opri execuţia unui fişier de comenzi prin tastarea lui CTRL/ /Z, caz în care sistemul de operare întreabă dacă se doreşte terminarea fişierului de comenzi.

Deci, pentru utilizarea fişierelor de comenzi sînt necesare trei etape:

1 — scrierea programului cu un editor sau cu comanda *copy*;

2 — crearea unui fişier cu extensia .BAT, în care să fie introduse comenzile programului;

3 — execuţia programului prin simpla tastare a numelui fişierului de comenzi.

4.6.1. Fişierul AUTOEXEC.BAT

Fişierul AUTOEXEC.BAT permite executarea automată a unor programe cu comenzi MS-DOS, şi caută pe disc în directorul rădăcină fişierul AUTOEXEC.BAT. Dacă este găsit, se vor executa comenzile memorate în el. Aici pot fi introduse o serie de comenzi de iniţializare, după dorinţa utilizatorului. Este recomandabil să fie memorate aici şi comenzile *time* şi *date*, deoarece în cazul existenţei unui fişier AUTOEXEC.BAT sistemul de operare nu mai întreabă utilizatorul asupra datei şi orei, iar dacă în fişier lipsesc aceste comenzi nu mai pot fi gestionate corect informaţiile din directoare. În figura 4.12 este ilustrat modul în care sistemul de operare MS-DOS foloseşte fişierul de comenzi AUTOEXEC.BAT.

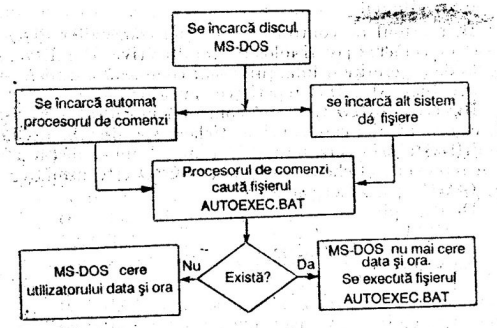


Fig. 4.12. Utilizarea fişierului AUTOEXEC. BAT

4.6.2. Utilizarea parametrilor în fişierul de comenzi

Adeseori dorim să executăm un fişier de comenzi cu diferite seturi de date. În acest scop, în interiorul unui fişier de comenzi se pot include parametrii formali care vor fi înlocuiţi cu valorile introduse la lansarea în execuţie a fişierului. Aceşti parametri sînt poziţionaţi în linia de comandă şi se identifică prin numele %0,%1, ..., %9 (parametrul %0 este întotdeauna numele fişierului de comenzi, memorat sub numele F1.BAT):

— *del* %3.PAS
— *copy* %1. pas + %2. pas %3.pas
— *print* %3.pas

Cînd se lansează în execuţie fişierul, trebuie furnizaţi cei trei parametri necesari, ca de exemplu:

c) F1 prog 1 prog 2 prog 3

În acest caz se şterge fişierul prog3.pas (%3), apoi se vor concatena prog 1.pas (%1) şi prog2.pas (%2). Într-un alt fişier prog 3.pas, care va fi apoi listat la imprimantă. După cum se observă parametrul %1 va conţine şirul „prog1", %2 şirul „prog2", iar 3% şirul „prog3". În mod uzual un program poate folosi cel mult 10 parametri. Dar cu directiva *shift* se poate extinde, dacă se doreşte, numărul parametrilor unui fişier de comenzi.

4.6.3. Directive

În procesul de control al execuţiei comenzilor dintr-un fişier de comenzi pot fi folosite trei directive. Rolul lor este acela de a permite o înlănţuire mai complexă a comenzilor, cu eventuale structuri repetitive sau alternative.

Directiva GOTO transferă controlul în cadrul unui fişier de comenzi liniei care conţine eticheta specificată. Eticheta se defineşte prin caracterul (:), urmat de un şir de caractere care reprezintă eticheta. Sintaxa lui GOTO este următoarea:

GOTO [:] ⟨ eticheta ⟩

De exemplu:

```
: input
rem început de program
del * obj
  .
  .
  .
goto început
```


Directiva ECHO permite inhibarea afişării pe display a comenzii MS-DOS în timpul execuţiei lor. Ea nu are încă efect asupra unor mesaje pe care le produce şi le afişează aceste comenzi.

Sintaxa: ECHO [on: off: ⟨ mesaj ⟩]

Cînd se foloseşte opţiunea *on*, afişarea comenzilor este activă. Cînd se specifică ECHO ⟨mesaj ⟩, mesajul va fi afişat pe consolă indiferent de starea directivei ECHO (*on* sau *off*).

Directiva CALL permite apelarea unui fişier de comenzi din orice linie a altui fişier. După execuţia fişierului apelat se va reveni în fişierul origine, la următoarea linie după CALL. În acest caz se permite tratarea unui fişier de comenzi ca o singură comandă.

Sintaxa: CALL [⟨d : ⟩] [calc] ⟨ nume-fişier ⟩

Directiva FOR permite prelucrarea iterativă a unor comenzi dintr-un fişier de comenzi.

Sintaxa: FOR %% ⟨ variabila ⟩ IN (⟨ mulţime ⟩) DO ⟨ comanda ⟩

Se va evalua ⟨ comanda ⟩ prin înlocuirea %% ⟨variabila ⟩ pe rînd cu fiecare valoare din ⟨ mulţime ⟩. După fiecare înlocuire ⟨ comanda ⟩ va fi executată. Dacă un element al mulţimii este un nume de fişier construit cu metacaracterele * sau ? , atunci %% ⟨variabilă ⟩ se va înlocui cu toate numele de fişier care se pot genera pe baza acestor caractere.

De exemplu, for %%f in (*.pas) do type %%f va tipări la consolă toate fişierele cu extensia .pas din directorul curent.

Directiva IF permite execuţia condiţionată a unei comenzi din fişierul de comenzi.

Sintaxa: IF [NOT] ⟨ condiţie ⟩ ⟨ comanda ⟩

unde: ⟨ condiţie ⟩ poate fi:

— ⟨şir1 ⟩ = ⟨şir2 ⟩; condiţia devine adevărată dacă cele două şiruri sînt egale;
— EXIST [⟨d1: ⟩] [cale] ⟨fişier ⟩ [.ext] condiţia devine adevărată dacă fişierul este găsit;
— ERRORLEVEL ⟨număr ⟩, condiţia devine adevărată dacă programul precedent executat are un cod de retur de valoare mai mare sau egal cu ⟨număr ⟩.

De exemplu:

IF NOT EXIST A : AUTOEXEC.BAT ECHO

Nu există fișier, AUTOEXEC va căuta pe discul A: fișierul AUTOEXEC.BAT și dacă nu există tipărește un mesaj la consolă.

Directiva REM permite afișarea unor comentarii în timpul execuției comenzilor dintr-un fișier de comenzi.

Sintaxa: REM ⟨comentariu ⟩

Directiva PAUSE suspendă execuția comenzilor dintr-un fișier de comenzi pînă la apăsarea oricărei taste.

Sintaxa: PAUSE [⟨comentariul ⟩]

De exemplu, *pause* schimbă discheta de pe unitatea A:, permite să se aștepte pînă ce utilizatorul pune o altă dischetă pe unitate și apoi apasă orice tastă.

Directiva SHIFT permite utilizarea a mai mult de 10 parametri în prelucrarea unui fișier de comenzi.

Sintaxa: SHIFT

Parametrii formali sînt numerotați de la %0 la %9. Prin execuția lui SHIFT toți parametrii sînt deplasați spre stînga cu o poziție: parametrul %0 va fi înlocuit cu parametrul %1 etc. Ca urmare, al 11-lea parametru se va introduce în %9 devenit liber și va putea fi folosit. Dacă se dorește utilizarea celui de-al 12-lea parametru se mai execută directiva SHIFT etc.

De exemplu, dacă parametrii au la un moment dat următoarele valori:

%0 = prog
%1 = 1
%2 = 2
:
:
:
%9 = 9

execuția directivei SHIFT va da următoarele rezultate:

%0 = 1
%1 = 2
:
:
:
%8 = 9
%9 = 10

4.7. EDITORUL STANDARD EDLIN

Editorul standard al sistemului MS-DOS este EDLIN, un editor orientat pe linii care permite:

— crearea unor fișiere text și salvarea lor pe disc;

— actualizarea unor fișiere text deja existente;

— ștergerea, editarea, inserarea sau afișarea unor linii de fișiere;

— căutarea unor linii conform unor șabloane.

EDLIN împarte textul în linii, fiecare linie avînd pînă la 253 caractere. EDLIN atribuie fiecărei linii cîte un număr care este afișat o dată cu linia pe ecran. Atunci cînd se fac inserări de noi linii sau ștergeri, numerele de linii sînt ajustate automat, astfel încît liniile să fie în permanență numerotate în secvență.

4.7.1. Lansarea lui EDLIN

Pentru a lansa editorul EDLIN se tastează:

edlin ⟨nume fișier ⟩

Pot exista două situații:

1) dacă fișierul indicat ⟨ nume fișier ⟩ există în director, acesta este încărcat în întregime în memoria internă, afișîndu-se mesajul:

End of input file

În acest moment utilizatorul poate afișa fișierul folosind comenzile EDLIN. Dacă fișierele indicate au o dimensiune prea mare ocupînd mai mult de 3/4 din memoria internă, se încarcă doar o porțiune din el; pe ecran apare prompterul

(*) și utilizatorul poate edita linii din porțiunea încărcată. Cu comenzile W (*write*) și A (*append*) poate comanda și încărcarea restului de fișiere;

2) dacă fișierul indicat ⟨ nume fișier ⟩ nu există în director, se va afișa mesajul:

New file

*

=

iar utilizatorul, cu ajutorul comenzii I (*insert*), va introduce liniile noului fișier.

4.7.2. Comenzi EDLIN

Cu ajutorul comenzilor EDLIN se comandă funcțiile de editare ce se doresc a se realiza în cadrul liniilor text. Pentru a referi liniile avem la dispoziție mai multe posibilități:

1) prin numărul absolut al liniei (între 1 și 65534);

2) prin metacaractere:

— metacaracterul # pentru linia de după ultima linie încărcată în memorie;

— metacaracterul . pentru linia curentă;

3) prin numere de linie relative la linia curentă.

Aici se pot folosi semnele (+, —) pentru a se indica dacă se merge înainte sau înapoi în fișier. De exemplu, —5, +5 indică un grup de linii cuprins între linia 5, înaintea liniei curente. și linia 5, după linia curentă. Vom prezenta în continuare principalele comenzi EDLIN, în ordine alfabetică:

Append (A) — permite adăugarea a *n* linii din fișierul disc în memorie, după linia curentă.

Sintaxa — [n]A

Dacă nu se specifică *n*, implicit se încarcă în memorie pînă la ocuparea a 3/4 din capacitatea ei. Această comandă se folosește în cazul editării unor fișiere mari. În aceste situații, la lansarea EDLIN se încarcă în memorie doar o parte din fișier. După editarea acestor linii, se vor scrie pe disc cu comanda W (*write*), iar apoi se va încărca cu comanda A o altă porțiune de fișier pentru a fi editată.

Copy (C) — permite copierea unui grup de linii dintr-un loc în altul al fișierului.

Sintaxa:

— ⟨ linia ⟩, ⟨ linia 2 ⟩[, ⟨ linia 3 ⟩]C

Se va copia grupul de linii cuprins între ⟨ linia 1 ⟩ și ⟨ linia 2 ⟩ înaintea liniei specificate de ⟨ linia 3 ⟩ (sau linia curentă dacă acest parametru lipsește). De exemplu, comanda —3, +3, 10 C determină copierea grupului de linii din jurul liniei curente (—3, +3) înaintea liniei 10.

Delete (D) — permite ștergerea unei linii sau grup de linii.

Sintaxa — [⟨ linie1 ⟩, ⟨ linie 2 ⟩] D

De exemplu, comanda: 1,D va antrena ștergerea tuturor liniilor de la începutul fișierului și pînă la linia curentă.

Edit — permite editarea unei linii

Sintaxa — ⟨ linie ⟩

Atunci cînd se tastează un număr de linie, EDLIN va afișa linia cerută și va aștepta introducerea unor comenzi date prin tastele de editare:

— DEC — salt peste un caracter din linia curentă. Cursorul nu este mutat;

— ESC — anulează linia curentă afișată;

— F1 sau — ⟩ — copiază un caracter din linia de intrare și îl afișează;

— F2 — copiază toate caracterele pînă la unul specificat (se mută cursorul după caracterul specificat);

— F3 — copiază tot restul liniei de intrare pe ecran;

— F4 — salt peste toate caracterele din linia de intrare pînă la un caracter specificat (se șterg toate caracterele pînă la cel specificat);

— INS — inserare de caractere în linie cu CTRL / BREAK.

De exemplu, vrem să editâm linia 5 din fișier:

*5

5 : ȘERBAN DAN GEORGE

5 : F2 D va determina mutarea cursorului în dreapta caracterului D

5 : ŞERBAN D DEL DEL F3 va şterge două caractere (A, N) şi va determina copierea restului liniei.
5 : ŞERBAN D GEORGE

În cazul în care cursorul este la sfîrşitul liniei, se intră automat în modul inserare. Cînd apare ENTER se salvează linia modificată şi se iese din editare.

End (E) — permite sfîrşitul editării. EDLIN va salva fişierul editat pe disc într-un fişier cu numele ⟨ nume fişier ⟩, iar vechea versiune va fi redenumită în ⟨ nume fişier ⟩.BAK.

În acest moment vor exista atît vechea versiune, cît şi versiunea modificată a fişierului care a constituit obiectul editării.

Insert (I) — permite inserarea unor linii imediat înainte de linia indicată.

Sintaxa — [⟨ linie ⟩] I

Odată intrat în modul inserare, se pot introduce oricite linii, fiecare terminîndu-se cu RETURN. Ieşirea din inserare se face cu CTRL/C.

List (L) — permite listarea unui grup de linii.

Sintaxa — [⟨ linia1 ⟩, ⟨ linia2 ⟩] L

De exemplu, comanda :, 25 L va face listarea tuturor liniilor între linia curentă şi linia 25 (inclusiv capetele intervalului).

Move line (M) — permite mutarea unui grup de linii dintr-un loc în altul cu ştergerea liniilor originale.

Sintaxa — ⟨ linie1 ⟩, ⟨ linie2 ⟩, [, ⟨ linie3 ⟩] M

Se va muta grupul ⟨ linie1 ⟩ , ⟨ linie2 ⟩ înaintea liniei ⟨ linie3 ⟩

De exemplu, +20,50M va determina mutarea grupului de linii cuprins între linia curentă, pînă la a 20-a după linia curentă, înaintea liniei 50.

Page (P) — permite listarea unui număr de linii specificat, cîte o pagină (23 linii) la un moment dat.

Sintaxa — [⟨linia1 ⟩, ⟨ linia2 ⟩] P

Spre deosebire de comanda L, aici linia curentă nu se modifică în urma listării.

Quit (Q) — permite ieşirea din EDLIN, fără salvarea fişierului editat.

Replace (R) — face înlocuirea tuturor aparitiilor unui şir de caractere într-un grup de linii, cu un alt şir de caractere.

Sintaxa — [⟨linie1 ⟩, ⟨ linie2 ⟩] [?]R ⟨ şir-vechi⟩ ⟨CTRL / Z ⟩ ⟨şir-nou ⟩

Caracterul ? folosit în comandă permite ca înlocuirea să se oprească de fiecare dată cînd într-o linie se găseşte ⟨ şir-vechi ⟩. Se va afişa linia modificată cu ⟨ şir-nou ⟩ şi se va cere confirmarea din partea utilizatorului.

De exemplu, comanda:

1, # R while ⟨ CTRL / Z ⟩ repeat

va înlocui în tot fişierul şirul „while" în şirul „repeat".

Search (s) — permite căutarea unei linii, într-un grup indicat, care să conţină un grup de caractere.

Sintaxa — [⟨ linie1 ⟩, ⟨linie2 ⟩] [?] S ⟨ şir ⟩

Linia găsită va deveni linie curentă.

Transfer (T) — transferă (inserează) conţinutul unui fişier specificat în fişierul curent de editare.

Sintaxa — [⟨ linie ⟩] T ⟨ fişier ⟩

Parametrul ⟨ linie ⟩, indică înaintea cărei linii să se insereze ⟨ fişier ⟩.

write (w) — permite scrierea unui anumit număr de linii pe disc.

Sintaxa — [⟨ număr ⟩]W

Implicit, sînt scrise 1/4 linii din liniile existente în memoria internă. Liniile sînt scrise începînd cu linia 1. Comanda este folosită în cazul editării fişierelor foarte mari, în conjuncţie cu comanda A (*append*).

4.8. CONFIGURAREA SISTEMULUI DE OPERARE

Configuraţia sistemului de operare trebuie să fie flexibilă şi să poată fi modificată în funcţie de necesităţi şi de echipamentele conectate la sistem. În acest scop, există un fişier numit CONFIG.SYS, care conţine anumite comenzi pentru definirea structurii sistemului de operare. La lansare MS-DOS caută în directorul rădăcină de pe discul sistem acest fişier; în cazul în care este găsit, se interpretează co-

menzile. Dacă fişierul nu este găsit, configurarea sistemului de operare se face cu valorile implicite ale comenzilor. Astfel, se poate modifica structura sistemului de operare cu un efort minim (de exemplu, se pot adăuga noi drivere). Modificarea fişierului CONFIG.SYS se poate face cu EDLIN sau cu oricare alt editor. Schimbările devin efective numai la o nouă iniţializare a sistemului de operare. Vom indica în continuare sintaxa şi semantica comenzilor utilizate în acest fişier.

4.8.1. Comanda BREAK

Funcţia — permite setarea verificării caracterului CTRL/C (CTRL/BREAK).

Sintaxa — BREAK = [on : off]

În mod normal sistemul de operare verifică introducerea acestui caracter numai în timpul operaţiilor de I/E cu echipamentul standard (display). Prin această comandă se activează verificarea şi la alte echipamente (de exemplu, disc), la fiecare apel al funcţiilor sistem. Modificarea ulterioară a funcţiei de verificare se poate face şi cu comanda MS-DOS, BREAK.

4.8.2. Comanda BUFFERS

Funcţia — permite setarea numărului de buffere disc pe care le alocă în memorie MS-DOS.

Sintaxa — BUFFERS = ⟨ număr ⟩

Un buffer disc este un bloc de memorie de 528 octeţi, unde MS-DOS păstrează datele citite / scrise de pe / pe disc, atunci cînd acestea nu sînt un multiplu al dimensiunii sectorului. Sistemul de operare va reţine în buffer cele mai recente zone de date utilizate. De aceea se recomandă un număr mai mare de buffere pentru a se accelera lucrul cu fişierul ori de cîte ori se primeşte o comandă de **I/E** a unei înregistrări care nu este multiplu exact al dimensiunii sectorului, se testează întîi dacă sectorul care conţine înregistrarea nu este deja într-un buffer. Dacă nu este se va citi sectorul; dacă se găseşte, atunci are loc un transfer simplu al înregistrării din buffer în zona de memorie a programului. Valoarea implicită a numărului de buffere este 2; în aplicaţii însă un mare trafic cu discul (ca de exemplu, procesare de texte) cele mai bune performanţe se obţin pentru numere cuprinse între 10 şi 20. Cînd sistemul de fişiere conţine multe subdirectoare, este indicat să se utilizeze un număr cuprins între 10 şi 20.

4.8.3. Comanda COUNTRY

Funcţia — Stabileşte convenţia (formatul) în care să furnizeze MS-DOS data şi ora. Aceste convenţii diferă de la o ţară la alta.

Sintaxa — COUNTRY = ⟨ x ⟩

De exemplu, valoarea 001 indică convenţia SUA, 003 Franţa etc.

4.8.4. Comanda DEVICE

Funcţia — instalează un driver de intrare-ieşire într-un anumit director.

Sintaxa — DEVICE = [⟨d: ⟩][⟨ cale ⟩] ⟨ nume ⟩ [. ⟨ extensie ⟩]

În timpul încărcării, sistemul de operare încarcă aceste fişiere în memorie ca o extensie a sa şi dă controlul, cînd este nevoie, acestor drivere. Driverele standard, încărcate de MS-DOS, sînt cele pentru display, tastatură, imprimantă, disc şi ceas. Pentru aceasta nu este nevoie de comanda DEVICE. Dacă utilizatorul a scris un driver de echipament care se doreşte să fie încărcat o dată cu lansarea sistemului, aceasta trebuie inclus în fişierul CONFIG.SYS printr-o comandă DEVICE. Dacă se înlocuieşte un driver standard se pot folosi următoarele programe de pe discul sistem:

a) *Fişierul ANSI.SYS* extinde facilităţile claviaturii şi ecranului la „setul extins„. Se înlocuiesc driverele standard pentru display şi tastatură;

b) *Fişierul DRIVER.SYS* permite introducerea unui driver pentru echipamentede tip bloc. Se crecază astfel echipamente logice de tip disc, care sînt referite prin litere de unitate, ca orice disc din configuraţie. Formatul comenzii în acest caz este:

DEVICE = DRIVER..SYS / D: ⟨ n1⟩ [/ T: ⟨ n2 ⟩] [/s : ⟨ n3 ⟩] [/H: ⟨n4 ⟩][/F: ⟨n5⟩] [/N]

unde/D :⟨ N1 ⟩— specifică numărul de unitate (0-255). Valoarea o specifică prin unitatea de disc flexibil referită cu litera A etc. Pentru a specifica prima unitate de disc Winchester se va folosi valoarea 128;

/T : ⟨N2 ⟩ — specifică numărul de piste pe față (1-999)

/S : ⟨N3 ⟩ — specifică numărul de sectoare pe pista (1-99);

/H:e ⟨N4 ⟩ — specifică numărul de capete de citire / scriere;

/F : ⟨N5 ⟩ — specifică tipul unității (0 pentru 360K0, 2 pentru 720K0 și altele);

/N — specifică faptul că discul este amovibil. (discurile Winchester sînt de acest tip).

c) *Fișierul VDISK.SYS* permite gestiunea unui disc virtual în memorie. Este vorba de un disc simulat în memorie, ca o unitate de disc flexibilă, cu următoarele proprietăți

— crește viteza de lucru cu „discul";

— conținutul discului virtual se pierde la reîncărcarea sistemului;

— se adresează printr-o literă;

— se pot instala mai multe discuri virtuale, în funcție de capacitatea memoriei;

— pentru fiecare „disc" se pot specifica dimensiunea, dimensiunea sectorului și numărul de intrări în director;

— discurile virtuale nu pot fi formatate. Odată instalat el este deja formatat. Formatul comenzii în acest caz este: DEVICE = VDISC.SYS[⟨comentariu ⟩][⟨n 1 ⟩] [⟨comentariu ⟩][⟨n2 ⟩][comentariu][⟨n3 ⟩]

unde ⟨comentariu ⟩ — reprezintă orice șir de caractere;

⟨n1 ⟩ — dimensiunea (în Kocteți) a discului virtual(între 1Ko și dimensiunea maximă a memoriei, minus 64 Kocteți);

⟨n2 ⟩ — specifică dimensiunea sectorului în octeți (128, 250 sau 512)

⟨n3 ⟩ — definește numărul de intrări în director (implicit 64).

De exemplu, comanda DEVICE = VDISC.SYS 180 256 64 permite crearea unui disc virtual de 180 Ko, 256 octeți / sector și cu 64 de intrări în director. La inițializarea sistemului de operare, acesta anunță litera care se atribuie discului virtual.

4.8.5. Comanda FCBS

Funcția — specifică numărul de FCB (*File Control Block*) structuri de date care pot fi deschise simultan.

Sintaxa — FCBS = ⟨x ⟩, ⟨y ⟩

unde

⟨x ⟩ — specifică numărul total de fișiere care pot fi deschise simultan (implicit 4);

⟨y ⟩ — specifică numărul de fișiere deschise simultan care nu pot fi închise simultan de MS-DOS, dacă programul încearcă să deschidă simultan mai mult de *n* fișiere.

4.8.6. Comanda FILES

Funcția — setează numărul de fișiere deschise compatibile cu sistemul de operare XENIX (de tipul UNIX) la care pot avea acces apelurile sistem MS-DOS.

Sintaxa — FILES = ⟨număr ⟩

4.8.7. Comanda LASTDRIVE

Funcția — setează numărul maxim de unități disc la care există posibilitatea de acces.

Sintaxa — **LASTDRIVE** = ⟨x ⟩.

unde:

⟨x ⟩ — reprezintă o literă între A și Z (implicit E).

4.8.8. Comanda SHELL

Funcția — permite specificarea unui alt interpretor de comenzi în locul lui COMMAND.COM.

Sintaxa — SHELL [⟨d : ⟩][⟨cale ⟩] ⟨nume ⟩ [. ⟨ext ⟩]

În acest fel programatorii, dacă doresc, pot dezvolta propriul lor interpretor de comenzi. Vor trebui însă să trateze întreruperile 22, 23 și 24 hexa, precum și citirea și execuția comenzilor. În acest caz nu mai sînt disponibile comenzile interne, interpretorul fișierelor de comenzi indirecte și încărcătorul de program. Ele vor trebui copiate în noul interpretor creat.

4.8.9. Comanda STACKS

Funcția — permite mărirea dimensiunii implicite a stivei.

Sintaxa — STACKS = ⟨n1⟩, ⟨n2⟩.

unde ⟨n1⟩ — reprezintă numărul de zone de stivă;

⟨n2⟩ — în octeți a fiecărei zone.

La fiecare întrerupere hardware apărută, sistemul de operare alocă o zonă de stive. După tratarea întreruperii, zona respectivă se eliberează. Dacă nu se specifică comanda STACKS, înseamnă că nu se dorește utilizarea unei stive dinamice suplimentare, ci numai a stivei interne.

BIBLIOGRAFIE SELECTIVĂ

1. Bourne, S.R.: „The UNIX Operating System", Addison Wesley, 1984.
2. Brown, P.: „Starting with UNIX", Addison Wesley, 1984.
3. Bynon, D.W., Shannon ,T.C.: „Introduction to VAX/VMS", Professional Press Inc. Spring House, 1987.
4. Christian, K.: „The UNIX Operating System", John Wiley, 1983.
5. Coffman, E.G., Elphick, M.J., Shoshani, A.: „System Deadlock". Computing Surveys, iunie, 1971.
6. Deitel, H.M.: „An Introduction to Operating Systems", Addison Wesley, 1984.
7. Denning, D.: „Cryptography and Data Security", Addison Wesley, 1984.
8. Finkel, R.A.: „An Operating Systems Vade Mecum", Prentice-Hall, 1986.
9. Fites, P., Jhonston, P., Kratz, M.: „The Computer Virus Crisis", Van Nostrand Reinhold, New York, 1989.
10. Foxley, E.: „UNIX for Super-User", Addison Wesley, 1985.
11. Grosshaus, D.: „File Systems Design and Implementation", Englewood Cliffs, Prentice-Hall, 1986.
12. Highland, H.: „Computer Virus Handbook", Elsevier Advanced Technologies, 1990.
13. Kernighan, B.W., Pike, R.: „The UNIX Programming Environment", Prentice-Hall, 1984.
14. Kernighan, B.W., Ritchie, D.M.: „The C Programming Language", Prentice-Hall, 1988.
15. Kochan, S.G., Wood, P.H.: „UNIX System Security", Hasbrouck Heights, 1985.
16. Mc Gilton, H., Morgan, R.: „Introducing the UNIX System", Mc Grow Hill, 1983
17. Pasternack, J.: „Exploring the UNIX Environment", Boutom Books, 1985
18. Peterson, J.L., Silberschatz, A.: „Operating System Concepts", Addison Wesley, 1986.
19. Tanenbaum, A.S.: „Operating Systems-Design and Implementation", Prentice-Hall, 1988.
20. *** Documentație de firmă Microsoft pentru sistemul de operare MS-DOS.
21. *** Documentație de firmă AT & T Bell Laboratories pentru sistemul de operare UNIX.
22. *** Documentație de firmă Digital Equipment pentru sistemul de operare VMS.

CUPRINS

Cuvînt înainte 5

Capitolul 1
STRUCTURA SISTEMELOR DE OPERARE

1.1. INTERACȚIUNEA HARDWARE-SOFTWARE 9
1.2. CE ESTE UN SISTEM DE OPERARE 11
1.3. CONCEPTE ALE SISTEMULUI DE OPERARE 13
1.4. TIPURI DE SISTEME DE OPERARE 16
1.5. GESTIUNEA PROCESELOR 23
1.5.1. Noțiunea de proces 23
1.5.2. Cooperarea proceselor 27
1.5.2.1. Condiții de concurență 27
1.5.2.2. Secțiuni critice 28
1.5.2.3. Excluziunea mutuală prin așteptare — ocupare („busy—waiting") 29
1.5.2.4. Excluziunea mutuală prin așteptare — activare („sleep and Wakeup") 32
1.5.2.5. Excluziune mutuală prin semafoare 34
1.5.2.6. Contoare eveniment 37
1.5.2.7. Sincronizare prin variabile comune 39
1.5.2.8. Sincronizarea prin monitoare 40
1.5.2.9. Sincronizare prin mesaje 43
1.5.2.10. Sincronizarea prin randez-vous 45
1.5.3. Planificarea proceselor 48
1.5.3.1. Planificarea circulară 48
1.5.3.2. Planificator după priorități 49
1.5.3.3. Planificarea pe fire de așteptare multiple . . . 50
1.5.3.4. Planificare uniformă 50
1.5.3.5. Planificarea pe 2 nivele 50
1.6. GESTIUNEA INTRĂRILOR/IEȘIRILOR 51
1.6.1. Principii hardware de I/E 51
1.6.2. Principii software de I/E 53
1.6.3. Blocarea proceselor la resurse 55
1.6.3.1. Modelarea blocărilor 55
1.6.3.2. Ignorarea blocărilor 57
1.6.3.3. Detecția blocărilor 57
1.6.3.4. Prevenirea blocărilor 57
1.6.3.5. Evitarea blocărilor 59

1.7. SISTEMUL DE FIȘIERE 62
1.7.1. Conceptul de fișier 62
1.7.2. Fișiere bandă și disc 64
1.7.3. Directoare 66
1.7.4. Alocarea spațiului disc 67
1.7.4.1. Dimensiunea blocurilor 68
1.7.4.2. Evidența spațiului liber 69
1.7.4.3. Evidența spațiului alocat 69
1.7.4.3.1. Alocarea contiguă 69
1.7.4.3.2. Alocarea înlănțuită 70
1.7.4.3.3. Alocarea indexată 71
1.8. GESTIUNEA MEMORIEI 72
1.8.1. Organizarea memoriei 72
1.8.2. Monoprogramarea 73
1.8.3. Multiprogramarea cu partiții fixe 74
1.8.4. Multiprogramarea cu partiții variabile 76
1.8.4.1. Gestiunea memoriei prin harta de alocare . 77
1.8.4.2. Gestiunea memoriei prin liste înlănțuite . . 78
1.8.4.3. Gestiunea memoriei prin fragmente 78
1.8.4.4. Strategii de alocare 81
1.8.5. Multiprogramarea prin evacuare/reîncărcare 81
1.8.6. Memoria virtuală 82
1.8.6.1. Paginarea 83
1.8.6.2. Segmentarea 85
1.8.6.3. Segmentarea și paginarea 88
1.8.7. Strategii de înlocuire a paginilor 90
1.8.7.1. Algoritmul optimal 90
1.8.7.2. Algoritmul aleatoriu 91
1.8.7.3. Algoritmul FIFO 91
1.8.7.4. Algoritmul LRU 93
1.8.7.5. Algoritmul LFU 93
1.8.7.6. Algoritmul NUR 94
1.8.7.7. Dimensiunea paginilor 94

Capitolul 2
SISTEMUL DE OPERARE UNIX

2.1. SCURT ISTORIC 98
2.2. CARACTERISTICI ALE SISTEMULUI UNIX 102
2.3. STRUCTURA SISTEMULUI UNIX 104
2.4. SISTEMUL DE FIȘIERE 105
2.4.1. Fișiere ordinare 105
2.4.2. Fișiere director 106
2.4.3. Fișiere speciale 109
2.4.4. Mecanismul de protecție a fișierelor 110
2.5. INTERFAȚA UTILIZATOR 112
2.5.1. Sintaxa comenzilor Shell 112
2.5.2. Procese paralele 114
2.5.3. Fișiere standard de intrare și ieșire 115

2.5.4. Conectarea programelor prin pipe 117
2.5.5. Numele fişierelor 118
2.5.6. Comenzi Shell 119
2.5.6.1. Comenzi pentru gestiunea directoarelor . . . 119
2.5.6.2. Comenzi pentru fişiere text 122
2.5.6.3. Comenzi pentru gestiunea fişierelor 127
2.5.6.4. Comenzi pentru gestiunea volumelor 133
2.5.6.5. Comenzi pentru gestiunea proceselor 134
2.5.6.6. Comenzi informaţionale şi pentru terminale . 137
2.6. LIMBAJUL DE PROGRAMARE SHELL 140
2.6.1. Execuţia programelor Shell 140
2.6.2. Variabile Shell 141
2.6.3. Proceduri Shell 143
2.6.4. Instrucţiuni de control 143
2.6.4.1. Decizia 144
2.6.4.2. Ciclul 145
2.6.4.3. Comenzi speciale 146
2.6.4.4. Substituţia comenzilor 146
2.6.5. Exemple de programe Shell 147
2.7. INTERFAŢA PROGRAMATOR 152
2.7.1. Editorul de texte ed 153
2.7.2. Editorul de texte vi 157
2.7.3. Limbajul C 160
2.7.3.1. Caracteristici ale limbajului C 160
2.7.3.2. Structura programelor C 163
2.7.3.3. Tipuri de date de bază 163
2.7.3.4. Constante 164
2.7.3.5. Variabile 164
2.7.3.6. Expresii 166
2.7.3.7. Tipuri de date complexe 167
2.7.3.8. Instrucţiuni pentru controlul fluxului . . . 169
2.7.3.9. Funcţii 171
2.7.3.10. Preprocesorul C 171
2.7.3.11. Intrări/ieşiri 172
2.7.3.12. Compilatorul C 172
2.7.3.13. Exemple 174
2.7.4. Pascal Berkely 175
2.7.5. Fortan 77 177
2.7.6. Ratfor 180
2.8. FACILITĂŢI DE PROTECŢIE CRIPTOGRAFICĂ LA UNIX 183
2.8.1. Facilităţi criptografice în versiunea 6 183
2.8.2. Facilităţi criptografice în versiunea 7 184
2.8.2.1. Protecţia parolelor 184
2.8.2.2. Protecţia fişierelor 190
2.8.2.3. Poşta electronică secretă 191
2.9. NUCLEUL UNIX 195
2.9.1. Gestiunea proceselor 196
2.9.1.1. Planificarea proceselor 197
2.9.1.2. Structuri de date pentru gestiunea proceselor 198

2.9.1.3. Crearea dinamică a proceselor 201
2.9.1.4. Iniţierea proceselor 204
2.9.2. Gestiunea fişierelor 206
2.9.3. Gestiunea perifericelor 212

Capitolul 3

SISTEMUL DE OPERARE VMS

3.1. INTRODUCERE ÎN VAX ŞI VMS 214
3.2. ORGANIZAREA SISTEMULUI DE OPERARE VMS . . . 220
3.2.1. Gestiunea memoriei 220
3.2.2. Încărcarea proceselor 222
3.2.3. Gestiunea intrărilor/ieşirilor 225
3.2.4. Sincronizarea şi cooperarea proceselor 231
3.3. FIŞIERE ŞI DIRECTOARE 232
3.3.1. Specificarea unui fişier 233
3.3.2. Tipuri de fişiere 236
3.4. INTERFAŢA UTILIZATOR. LIMBAJUL DCL 239
3.4.1. Iniţierea unei sesiuni 239
3.4.2. Limbajul de comandă DCL 240
3.4.2.1. Comenzi DCL referitoare la fişiere 243
3.4.2.2. Comenzi DCL referitoare la directoare . . . 251
3.4.2.3. Comenzi DCL referitoare la volume 254
3.4.2.4. Comenzi DCL referitoare la nume logice . . 255
3.4.2.5. Comenzi DCL informaţionale şi de control al contextului utilizator 259
3.4.2.6. Comenzi DCL referitoare la procese 268
3.4.2.7. Comenzi DCL referitoare la securitate . . . 271
3.5. EDITORUL DE TEXTE EDT 275
3.5.1. Apelarea EDT 276
3.5.2. Moduri de lucru cu EDT 276
3.5.2.1. Modul linie 277
3.5.2.2. Modul caracter 278
3.5.3. Generarea manualului EDT 281
3.5.4. Fişierul jurnal 282
3.5.5. Editorul orientat pe sintaxa LSEDIT 282
3.6. PROCEDURI DE COMANDĂ 284
3.6.1. Simboluri 284
3.6.2. Reprezentarea datelor şi expresiilor 286
3.6.3. Funcţii lexicale 286
3.6.4. Crearea procedurilor de comandă 289
3.6.5. Apelarea procedurilor de comandă 290
3.6.6. Folosirea simbolurilor şi funcţiilor lexicale. Proceduri de comandă 291
3.6.7. Intrări/ieşiri în proceduri de comandă 292
3.6.8. Utilizarea fişierului în proceduri de comandă . . . 293
3.6.9. Instrucţiuni pentru controlul secvenţei de execuţie . . 293
3.6.10. Exemple de proceduri de comandă 296

3.7. PROGRAME UTILITARE 300
3.7.1. Utilitarul HELP 300
3.7.2. Utilitarul BACKUP 301
3.7.3. Utilitarul LIBRARY 304
3.7.4. Utilitarul SORT 305
3.7.5. Utilitarul MAIL 306
3.7.6. Utilitarul PHONE 308
3.7.7. Alte utilitare disponibile sub VMS 309

Capitolul 4
SISTEMUL DE OPERARE MS–DOS

4.1. STRUCTURA SISTEMULUI MS–DOS 311
4.2. ORGANIZAREA DISCULUI LA MS–DOS 316
4.2.1. Organizarea discului flexibil 316
4.2.2. Utilizarea discului Winchester 321
4.3. SISTEMUL DE FIŞIERE LA MS–DOS 323
4.3.1. Specificarea fişierelor 325
4.4. FORMATUL PROGRAMELOR EXECUTABILE 327
4.5. INTERFAŢA UTILIZATOR 328
4.5.1. Redirectarea fişierelor standard 330
4.5.2. Comenzi MS–DOS 331
4.5.2.1. Comenzi cu privire la directoare 331
4.5.2.2. Comenzi cu privire la volume 334
4.5.2.3. Comenzi cu privire la fişiere 337
4.5.2.4. Comenzi filtru 343
4.5.2.5. Comenzi pentru definirea unor opţiuni . . . 344
4.5.2.6. Comenzi informaţionale 347
4.6. FIŞIERE DE COMENZI 348
4.6.1. Fişierul AUTOEXEC. BAT 349
4.6.2. Utilizarea parametrilor în fişierul de comenzi 350
4.6.3. Directive 350
4.7. EDITORUL STANDARD EDLIN 353
4.7.1. Lansarea lui EDLIN 353
4.7.2. Comenzi EDLIN 354
4.8. CONFIGURAREA SISTEMULUI DE OPERARE 357
4.8.1. Comanda BREAK 358
4.8.2. Comanda BUFFERS 358
4.8.3. Comanda COUNTRY 359
4.8.4. Comanda DEVICE 359
4.8.5. Comanda FCBS 361
4.8.6. Comanda FILES 361
4.8.7. Comanda LASTDRIVE 361
4.8.8. Comanda SHELL 361
4.8.9. Comanda STACKS 362
BIBLIOGRAFIE SELECTIVĂ 363